GNU Emacs Lisp Reference Manual

For Emacs Version 29.1.90

**by Bil Lewis, Dan LaLiberte, Richard Stallman,
the GNU Manual Group, et al.**

This is the *GNU Emacs Lisp Reference Manual* corresponding to Emacs version 29.1.90.

Copyright © 1990–1996, 1998–2024 Free Software Foundation, Inc.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with the Invariant Sections being "GNU General Public License," with the Front-Cover Texts being "A GNU Manual," and with the Back-Cover Texts as in (a) below. A copy of the license is included in the section entitled "GNU Free Documentation License."

(a) The FSF's Back-Cover Text is: "You have the freedom to copy and modify this GNU manual. Buying copies from the FSF supports it in developing GNU and promoting software freedom."

Published by the Free Software Foundation
51 Franklin St, Fifth Floor
Boston, MA 02110-1301
USA
ISBN 1-882114-74-4

Cover art by Etienne Suvasa.

Short Contents

1 Introduction 1
2 Lisp Data Types 8
3 Numbers 38
4 Strings and Characters 54
5 Lists 77
6 Sequences, Arrays, and Vectors 102
7 Records 125
8 Hash Tables 127
9 Symbols 133
10 Evaluation 145
11 Control Structures 158
12 Variables 189
13 Functions 230
14 Macros 269
15 Customization Settings 278
16 Loading 299
17 Byte Compilation 317
18 Compilation of Lisp to Native Code 328
19 Debugging Lisp Programs 334
20 Reading and Printing Lisp Objects 372
21 Minibuffers 386
22 Command Loop 424
23 Keymaps 481
24 Major and Minor Modes 525
25 Documentation 597
26 Files 610
27 Backups and Auto-Saving 661
28 Buffers 673
29 Windows 692
30 Frames 787
31 Positions 855
32 Markers 871
33 Text 880

34 Non-ASCII Characters 966
35 Searching and Matching 996
36 Syntax Tables 1025
37 Parsing Program Source 1042
38 Abbrevs and Abbrev Expansion 1069
39 Threads 1076
40 Processes 1081
41 Emacs Display 1133
42 Operating System Interface 1260
43 Preparing Lisp code for distribution 1308
A Emacs 28 Antinews 1314
B GNU Free Documentation License 1317
C GNU General Public License 1325
D Tips and Conventions 1336
E GNU Emacs Internals 1351
F Standard Errors 1395
G Standard Keymaps 1400
H Standard Hooks 1403
Index 1407

Table of Contents

1 Introduction 1

1.1 Caveats 1
1.2 Lisp History 1
1.3 Conventions 2
1.3.1 Some Terms 2
1.3.2 `nil` and `t` 2
1.3.3 Evaluation Notation 3
1.3.4 Printing Notation 3
1.3.5 Error Messages 3
1.3.6 Buffer Text Notation 4
1.3.7 Format of Descriptions 4
1.3.7.1 A Sample Function Description 4
1.3.7.2 A Sample Variable Description 6
1.4 Version Information 6
1.5 Acknowledgments 7

2 Lisp Data Types 8

2.1 Printed Representation and Read Syntax 8
2.2 Special Read Syntax 9
2.3 Comments 10
2.4 Programming Types 10
2.4.1 Integer Type 10
2.4.2 Floating-Point Type 11
2.4.3 Character Type 11
2.4.3.1 Basic Char Syntax 11
2.4.3.2 General Escape Syntax 12
2.4.3.3 Control-Character Syntax 13
2.4.3.4 Meta-Character Syntax 14
2.4.3.5 Other Character Modifier Bits 14
2.4.4 Symbol Type 14
2.4.5 Sequence Types 15
2.4.6 Cons Cell and List Types 16
2.4.6.1 Drawing Lists as Box Diagrams 16
2.4.6.2 Dotted Pair Notation 18
2.4.6.3 Association List Type 19
2.4.7 Array Type 19
2.4.8 String Type 20
2.4.8.1 Syntax for Strings 20
2.4.8.2 Non-ASCII Characters in Strings 20
2.4.8.3 Nonprinting Characters in Strings 21
2.4.8.4 Text Properties in Strings 21
2.4.9 Vector Type 22
2.4.10 Char-Table Type 22

2.4.11 Bool-Vector Type ... 23
2.4.12 Hash Table Type ... 23
2.4.13 Function Type ... 23
2.4.14 Macro Type ... 24
2.4.15 Primitive Function Type ... 24
2.4.16 Byte-Code Function Type ... 24
2.4.17 Record Type ... 25
2.4.18 Type Descriptors ... 25
2.4.19 Autoload Type ... 25
2.4.20 Finalizer Type ... 25
2.5 Editing Types ... 26
2.5.1 Buffer Type ... 26
2.5.2 Marker Type ... 26
2.5.3 Window Type ... 27
2.5.4 Frame Type ... 27
2.5.5 Terminal Type ... 27
2.5.6 Window Configuration Type ... 28
2.5.7 Frame Configuration Type ... 28
2.5.8 Process Type ... 28
2.5.9 Thread Type ... 28
2.5.10 Mutex Type ... 28
2.5.11 Condition Variable Type ... 29
2.5.12 Stream Type ... 29
2.5.13 Keymap Type ... 29
2.5.14 Overlay Type ... 29
2.5.15 Font Type ... 30
2.5.16 Xwidget Type ... 30
2.6 Read Syntax for Circular Objects ... 30
2.7 Type Predicates ... 31
2.8 Equality Predicates ... 34
2.9 Mutability ... 37

3 Numbers ... 38

3.1 Integer Basics ... 38
3.2 Floating-Point Basics ... 39
3.3 Type Predicates for Numbers ... 41
3.4 Comparison of Numbers ... 42
3.5 Numeric Conversions ... 43
3.6 Arithmetic Operations ... 45
3.7 Rounding Operations ... 48
3.8 Bitwise Operations on Integers ... 48
3.9 Standard Mathematical Functions ... 51
3.10 Random Numbers ... 52

4 Strings and Characters 54

4.1 String and Character Basics 54
4.2 Predicates for Strings 55
4.3 Creating Strings 55
4.4 Modifying Strings 59
4.5 Comparison of Characters and Strings 60
4.6 Conversion of Characters and Strings 65
4.7 Formatting Strings 66
4.8 Custom Format Strings 70
4.9 Case Conversion in Lisp 73
4.10 The Case Table 74

5 Lists 77

5.1 Lists and Cons Cells 77
5.2 Predicates on Lists 78
5.3 Accessing Elements of Lists 79
5.4 Building Cons Cells and Lists 82
5.5 Modifying List Variables 85
5.6 Modifying Existing List Structure 87
5.6.1 Altering List Elements with `setcar` 88
5.6.2 Altering the CDR of a List 89
5.6.3 Functions that Rearrange Lists 90
5.7 Using Lists as Sets 91
5.8 Association Lists 95
5.9 Property Lists 99
5.9.1 Property Lists and Association Lists 100
5.9.2 Property Lists Outside Symbols 100

6 Sequences, Arrays, and Vectors 102

6.1 Sequences 102
6.2 Arrays 115
6.3 Functions that Operate on Arrays 116
6.4 Vectors 117
6.5 Functions for Vectors 118
6.6 Char-Tables 119
6.7 Bool-vectors 121
6.8 Managing a Fixed-Size Ring of Objects 123

7 Records 125

7.1 Record Functions 125
7.2 Backward Compatibility 126

8 Hash Tables ... 127

8.1 Creating Hash Tables ... 127
8.2 Hash Table Access ... 129
8.3 Defining Hash Comparisons ... 130
8.4 Other Hash Table Functions ... 131

9 Symbols ... 133

9.1 Symbol Components ... 133
9.2 Defining Symbols ... 134
9.3 Creating and Interning Symbols ... 135
9.4 Symbol Properties ... 138
9.4.1 Accessing Symbol Properties ... 138
9.4.2 Standard Symbol Properties ... 139
9.5 Shorthands ... 142
9.5.1 Exceptions ... 143
9.6 Symbols with Position ... 143

10 Evaluation ... 145

10.1 Kinds of Forms ... 146
10.1.1 Self-Evaluating Forms ... 146
10.1.2 Symbol Forms ... 146
10.1.3 Classification of List Forms ... 147
10.1.4 Symbol Function Indirection ... 147
10.1.5 Evaluation of Function Forms ... 148
10.1.6 Lisp Macro Evaluation ... 149
10.1.7 Special Forms ... 149
10.1.8 Autoloading ... 150
10.2 Quoting ... 151
10.3 Backquote ... 151
10.4 Eval ... 152
10.5 Deferred and Lazy Evaluation ... 155

11 Control Structures ... 158

11.1 Sequencing ... 158
11.2 Conditionals ... 159
11.3 Constructs for Combining Conditions ... 162
11.4 Pattern-Matching Conditional ... 163
11.4.1 The `pcase` macro ... 164
11.4.2 Extending `pcase` ... 170
11.4.3 Backquote-Style Patterns ... 171
11.4.4 Destructuring with `pcase` Patterns ... 172
11.5 Iteration ... 174
11.6 Generators ... 175
11.7 Nonlocal Exits ... 177
11.7.1 Explicit Nonlocal Exits: `catch` and `throw` ... 177

11.7.2 Examples of `catch` and `throw` ... 178
11.7.3 Errors ... 179
11.7.3.1 How to Signal an Error ... 180
11.7.3.2 How Emacs Processes Errors ... 181
11.7.3.3 Writing Code to Handle Errors ... 182
11.7.3.4 Error Symbols and Condition Names ... 186
11.7.4 Cleaning Up from Nonlocal Exits ... 187

12 Variables ... 189

12.1 Global Variables ... 189
12.2 Variables that Never Change ... 189
12.3 Local Variables ... 190
12.4 When a Variable is Void ... 193
12.5 Defining Global Variables ... 194
12.6 Tips for Defining Variables Robustly ... 196
12.7 Accessing Variable Values ... 197
12.8 Setting Variable Values ... 198
12.9 Running a function when a variable is changed ... 200
12.9.1 Limitations ... 200
12.10 Scoping Rules for Variable Bindings ... 201
12.10.1 Dynamic Binding ... 201
12.10.2 Proper Use of Dynamic Binding ... 202
12.10.3 Lexical Binding ... 203
12.10.4 Using Lexical Binding ... 204
12.10.5 Converting to Lexical Binding ... 206
12.11 Buffer-Local Variables ... 207
12.11.1 Introduction to Buffer-Local Variables ... 207
12.11.2 Creating and Deleting Buffer-Local Bindings ... 208
12.11.3 The Default Value of a Buffer-Local Variable ... 211
12.12 File Local Variables ... 213
12.13 Directory Local Variables ... 217
12.14 Connection Local Variables ... 218
12.14.1 Connection Local Profiles ... 218
12.14.2 Applying Connection Local Variables ... 220
12.15 Variable Aliases ... 222
12.16 Variables with Restricted Values ... 223
12.17 Generalized Variables ... 224
12.17.1 The `setf` Macro ... 224
12.17.2 Defining new `setf` forms ... 226
12.18 Multisession Variables ... 227

13 Functions 230
13.1 What Is a Function? 230
13.2 Lambda Expressions 232
13.2.1 Components of a Lambda Expression 232
13.2.2 A Simple Lambda Expression Example 233
13.2.3 Features of Argument Lists 234
13.2.4 Documentation Strings of Functions 235
13.3 Naming a Function 236
13.4 Defining Functions 237
13.5 Calling Functions 239
13.6 Mapping Functions 241
13.7 Anonymous Functions 243
13.8 Generic Functions 245
13.9 Accessing Function Cell Contents 248
13.10 Closures 250
13.11 Open Closures 250
13.12 Advising Emacs Lisp Functions 252
13.12.1 Primitives to manipulate advices 253
13.12.2 Advising Named Functions 255
13.12.3 Ways to compose advice 257
13.12.4 Adapting code using the old defadvice 258
13.12.5 Advice and Byte Code 260
13.13 Declaring Functions Obsolete 260
13.14 Inline Functions 261
13.15 The `declare` Form 263
13.16 Telling the Compiler that a Function is Defined 265
13.17 Determining whether a Function is Safe to Call 266
13.18 Other Topics Related to Functions 267

14 Macros 269
14.1 A Simple Example of a Macro 269
14.2 Expansion of a Macro Call 269
14.3 Macros and Byte Compilation 271
14.4 Defining Macros 271
14.5 Common Problems Using Macros 272
14.5.1 Wrong Time 272
14.5.2 Evaluating Macro Arguments Repeatedly 272
14.5.3 Local Variables in Macro Expansions 274
14.5.4 Evaluating Macro Arguments in Expansion 274
14.5.5 How Many Times is the Macro Expanded? 275
14.6 Indenting Macros 276

15 Customization Settings 278
15.1 Common Item Keywords 278
15.2 Defining Customization Groups 280
15.3 Defining Customization Variables 281
15.4 Customization Types 285
15.4.1 Simple Types 286
15.4.2 Composite Types 287
15.4.3 Splicing into Lists 291
15.4.4 Type Keywords 292
15.4.5 Defining New Types 294
15.5 Applying Customizations 295
15.6 Custom Themes 296

16 Loading 299
16.1 How Programs Do Loading 299
16.2 Load Suffixes 301
16.3 Library Search 302
16.4 Loading Non-ASCII Characters 305
16.5 Autoload 305
16.5.1 Autoload by Prefix 308
16.5.2 When to Use Autoload 308
16.6 Repeated Loading 309
16.7 Features 310
16.8 Which File Defined a Certain Symbol 312
16.9 Unloading 314
16.10 Hooks for Loading 314
16.11 Emacs Dynamic Modules 315

17 Byte Compilation 317
17.1 Performance of Byte-Compiled Code 317
17.2 Byte-Compilation Functions 317
17.3 Documentation Strings and Compilation 320
17.4 Dynamic Loading of Individual Functions 320
17.5 Evaluation During Compilation 321
17.6 Compiler Errors 322
17.7 Byte-Code Function Objects 324
17.8 Disassembled Byte-Code 325

18 Compilation of Lisp to Native Code 328
18.1 Native-Compilation Functions 329
18.2 Native-Compilation Variables 331

19 Debugging Lisp Programs 334
19.1 The Lisp Debugger 334
19.1.1 Entering the Debugger on an Error 334
19.1.2 Debugging Redisplay Errors 336
19.1.3 Debugging Infinite Loops 337
19.1.4 Entering the Debugger on a Function Call 337
19.1.5 Entering the debugger when a variable is modified 338
19.1.6 Explicit Entry to the Debugger 338
19.1.7 Using the Debugger 339
19.1.8 Backtraces 339
19.1.9 Debugger Commands 340
19.1.10 Invoking the Debugger 341
19.1.11 Internals of the Debugger 343
19.2 Edebug 345
19.2.1 Using Edebug 346
19.2.2 Instrumenting for Edebug 347
19.2.3 Edebug Execution Modes 348
19.2.4 Jumping 349
19.2.5 Miscellaneous Edebug Commands 350
19.2.6 Breaks 351
19.2.6.1 Edebug Breakpoints 351
19.2.6.2 Global Break Condition 352
19.2.6.3 Source Breakpoints 352
19.2.7 Trapping Errors 352
19.2.8 Edebug Views 353
19.2.9 Evaluation 353
19.2.10 Evaluation List Buffer 354
19.2.11 Printing in Edebug 355
19.2.12 Trace Buffer 356
19.2.13 Coverage Testing 357
19.2.14 The Outside Context 358
19.2.14.1 Checking Whether to Stop 358
19.2.14.2 Edebug Display Update 358
19.2.14.3 Edebug Recursive Edit 359
19.2.15 Edebug and Macros 359
19.2.15.1 Instrumenting Macro Calls 359
19.2.15.2 Specification List 361
19.2.15.3 Backtracking in Specifications 364
19.2.15.4 Specification Examples 364
19.2.16 Edebug Options 365
19.3 Debugging Invalid Lisp Syntax 368
19.3.1 Excess Open Parentheses 369
19.3.2 Excess Close Parentheses 369
19.4 Test Coverage 370
19.5 Profiling 370

20 Reading and Printing Lisp Objects 372
20.1 Introduction to Reading and Printing 372
20.2 Input Streams 372
20.3 Input Functions 375
20.4 Output Streams 376
20.5 Output Functions 378
20.6 Variables Affecting Output 381
20.7 Overriding Output Variables 384

21 Minibuffers 386
21.1 Introduction to Minibuffers 386
21.2 Reading Text Strings with the Minibuffer 387
21.3 Reading Lisp Objects with the Minibuffer 392
21.4 Minibuffer History 393
21.5 Initial Input 395
21.6 Completion 396
21.6.1 Basic Completion Functions 396
21.6.2 Completion and the Minibuffer 399
21.6.3 Minibuffer Commands that Do Completion 401
21.6.4 High-Level Completion Functions 403
21.6.5 Reading File Names 405
21.6.6 Completion Variables 408
21.6.7 Programmed Completion 410
21.6.8 Completion in Ordinary Buffers 412
21.7 Yes-or-No Queries 414
21.8 Asking Multiple-Choice Questions 416
21.9 Reading a Password 418
21.10 Minibuffer Commands 418
21.11 Minibuffer Windows 419
21.12 Minibuffer Contents 421
21.13 Recursive Minibuffers 421
21.14 Inhibiting Interaction 422
21.15 Minibuffer Miscellany 422

22 Command Loop 424
22.1 Command Loop Overview 424
22.2 Defining Commands 425
22.2.1 Using `interactive` 425
22.2.2 Code Characters for `interactive` 428
22.2.3 Examples of Using `interactive` 431
22.2.4 Specifying Modes For Commands 431
22.2.5 Select among Command Alternatives 432
22.3 Interactive Call 433
22.4 Distinguish Interactive Calls 436
22.5 Information from the Command Loop 437

22.6 Adjusting Point After Commands 440
22.7 Input Events 441
22.7.1 Keyboard Events 441
22.7.2 Function Keys 442
22.7.3 Mouse Events 443
22.7.4 Click Events 443
22.7.5 Drag Events 446
22.7.6 Button-Down Events 447
22.7.7 Repeat Events 447
22.7.8 Motion Events 448
22.7.9 Touchscreen Events 449
22.7.10 Focus Events 450
22.7.11 Xwidget events 450
22.7.12 Miscellaneous System Events 451
22.7.13 Event Examples 455
22.7.14 Classifying Events 456
22.7.15 Accessing Mouse Events 457
22.7.16 Accessing Scroll Bar Events 460
22.7.17 Putting Keyboard Events in Strings 460
22.8 Reading Input 461
22.8.1 Key Sequence Input 462
22.8.2 Reading One Event 463
22.8.3 Modifying and Translating Input Events 467
22.8.4 Invoking the Input Method 468
22.8.5 Quoted Character Input 468
22.8.6 Miscellaneous Event Input Features 469
22.9 Special Events 471
22.10 Waiting for Elapsed Time or Input 471
22.11 Quitting 472
22.12 Prefix Command Arguments 474
22.13 Recursive Editing 476
22.14 Disabling Commands 478
22.15 Command History 478
22.16 Keyboard Macros 479

23 Keymaps 481

23.1 Key Sequences 481
23.2 Keymap Basics 482
23.3 Format of Keymaps 482
23.4 Creating Keymaps 484
23.5 Inheritance and Keymaps 488
23.6 Prefix Keys 489
23.7 Active Keymaps 490
23.8 Searching the Active Keymaps 491
23.9 Controlling the Active Keymaps 492
23.10 Key Lookup 495

23.11 Functions for Key Lookup ... 496
23.12 Changing Key Bindings ... 498
23.13 Low-Level Key Binding ... 502
23.14 Remapping Commands ... 504
23.15 Keymaps for Translating Sequences of Events ... 505
23.15.1 Interaction with normal keymaps ... 507
23.16 Commands for Binding Keys ... 507
23.17 Scanning Keymaps ... 508
23.18 Menu Keymaps ... 511
23.18.1 Defining Menus ... 511
23.18.1.1 Simple Menu Items ... 511
23.18.1.2 Extended Menu Items ... 512
23.18.1.3 Menu Separators ... 514
23.18.1.4 Alias Menu Items ... 515
23.18.2 Menus and the Mouse ... 515
23.18.3 Menus and the Keyboard ... 516
23.18.4 Menu Example ... 516
23.18.5 The Menu Bar ... 517
23.18.6 Tool bars ... 518
23.18.7 Modifying Menus ... 522
23.18.8 Easy Menu ... 522

24 Major and Minor Modes ... 525

24.1 Hooks ... 525
24.1.1 Running Hooks ... 526
24.1.2 Setting Hooks ... 526
24.2 Major Modes ... 527
24.2.1 Major Mode Conventions ... 529
24.2.2 How Emacs Chooses a Major Mode ... 532
24.2.3 Getting Help about a Major Mode ... 535
24.2.4 Defining Derived Modes ... 535
24.2.5 Basic Major Modes ... 537
24.2.6 Mode Hooks ... 538
24.2.7 Tabulated List mode ... 539
24.2.8 Generic Modes ... 542
24.2.9 Major Mode Examples ... 543
24.3 Minor Modes ... 544
24.3.1 Conventions for Writing Minor Modes ... 545
24.3.2 Keymaps and Minor Modes ... 547
24.3.3 Defining Minor Modes ... 547
24.4 Mode Line Format ... 551
24.4.1 Mode Line Basics ... 551
24.4.2 The Data Structure of the Mode Line ... 552
24.4.3 The Top Level of Mode Line Control ... 553
24.4.4 Variables Used in the Mode Line ... 555
24.4.5 %-Constructs in the Mode Line ... 557

24.4.6 Properties in the Mode Line ... 559
24.4.7 Window Header Lines ... 560
24.4.8 Emulating Mode Line Formatting ... 561
24.5 Imenu ... 562
24.6 Font Lock Mode ... 564
24.6.1 Font Lock Basics ... 565
24.6.2 Search-based Fontification ... 567
24.6.3 Customizing Search-Based Fontification ... 570
24.6.4 Other Font Lock Variables ... 572
24.6.5 Levels of Font Lock ... 574
24.6.6 Precalculated Fontification ... 574
24.6.7 Faces for Font Lock ... 574
24.6.8 Syntactic Font Lock ... 577
24.6.9 Multiline Font Lock Constructs ... 577
24.6.9.1 Font Lock Multiline ... 578
24.6.9.2 Region to Fontify after a Buffer Change ... 579
24.6.10 Parser-based Font Lock ... 580
24.7 Automatic Indentation of code ... 582
24.7.1 Simple Minded Indentation Engine ... 583
24.7.1.1 SMIE Setup and Features ... 583
24.7.1.2 Operator Precedence Grammars ... 584
24.7.1.3 Defining the Grammar of a Language ... 585
24.7.1.4 Defining Tokens ... 586
24.7.1.5 Living With a Weak Parser ... 587
24.7.1.6 Specifying Indentation Rules ... 588
24.7.1.7 Helper Functions for Indentation Rules ... 589
24.7.1.8 Sample Indentation Rules ... 590
24.7.1.9 Customizing Indentation ... 591
24.7.2 Parser-based Indentation ... 592
24.8 Desktop Save Mode ... 595

25 Documentation ... 597

25.1 Documentation Basics ... 597
25.2 Access to Documentation Strings ... 598
25.3 Substituting Key Bindings in Documentation ... 600
25.4 Text Quoting Style ... 602
25.5 Describing Characters for Help Messages ... 603
25.6 Help Functions ... 604
25.7 Documentation Groups ... 607

26 Files ... 610

26.1 Visiting Files ... 610
26.1.1 Functions for Visiting Files ... 610
26.1.2 Subroutines of Visiting ... 613
26.2 Saving Buffers ... 614
26.3 Reading from Files ... 616

26.4 Writing to Files ... 618
26.5 File Locks ... 619
26.6 Information about Files ... 621
26.6.1 Testing Accessibility ... 621
26.6.2 Distinguishing Kinds of Files ... 623
26.6.3 Truenames ... 625
26.6.4 File Attributes ... 626
26.6.5 Extended File Attributes ... 629
26.6.6 Locating Files in Standard Places ... 630
26.7 Changing File Names and Attributes ... 631
26.8 Files and Secondary Storage ... 636
26.9 File Names ... 636
26.9.1 File Name Components ... 637
26.9.2 Absolute and Relative File Names ... 639
26.9.3 Directory Names ... 640
26.9.4 Functions that Expand Filenames ... 641
26.9.5 Generating Unique File Names ... 644
26.9.6 File Name Completion ... 646
26.9.7 Standard File Names ... 647
26.10 Contents of Directories ... 648
26.11 Creating, Copying and Deleting Directories ... 650
26.12 Making Certain File Names "Magic" ... 651
26.13 File Format Conversion ... 656
26.13.1 Overview ... 656
26.13.2 Round-Trip Specification ... 656
26.13.3 Piecemeal Specification ... 659

27 Backups and Auto-Saving ... 661

27.1 Backup Files ... 661
27.1.1 Making Backup Files ... 661
27.1.2 Backup by Renaming or by Copying? ... 663
27.1.3 Making and Deleting Numbered Backup Files ... 664
27.1.4 Naming Backup Files ... 665
27.2 Auto-Saving ... 666
27.3 Reverting ... 669

28 Buffers ... 673

28.1 Buffer Basics ... 673
28.2 The Current Buffer ... 673
28.3 Buffer Names ... 676
28.4 Buffer File Name ... 677
28.5 Buffer Modification ... 679
28.6 Buffer Modification Time ... 680
28.7 Read-Only Buffers ... 682
28.8 The Buffer List ... 683
28.9 Creating Buffers ... 686

28.10 Killing Buffers ... 687
28.11 Indirect Buffers ... 689
28.12 Swapping Text Between Two Buffers ... 690
28.13 The Buffer Gap ... 691

29 Windows ... 692

29.1 Basic Concepts of Emacs Windows ... 692
29.2 Windows and Frames ... 694
29.3 Selecting Windows ... 698
29.4 Window Sizes ... 701
29.5 Resizing Windows ... 705
29.6 Preserving Window Sizes ... 709
29.7 Splitting Windows ... 710
29.8 Deleting Windows ... 713
29.9 Recombining Windows ... 714
29.10 Cyclic Ordering of Windows ... 719
29.11 Buffers and Windows ... 722
29.12 Switching to a Buffer in a Window ... 723
29.13 Displaying a Buffer in a Suitable Window ... 726
29.13.1 Choosing a Window for Displaying a Buffer ... 726
29.13.2 Action Functions for Buffer Display ... 728
29.13.3 Action Alists for Buffer Display ... 733
29.13.4 Additional Options for Displaying Buffers ... 738
29.13.5 Precedence of Action Functions ... 741
29.13.6 The Zen of Buffer Display ... 745
29.14 Window History ... 748
29.15 Dedicated Windows ... 751
29.16 Quitting Windows ... 752
29.17 Side Windows ... 754
29.17.1 Displaying Buffers in Side Windows ... 754
29.17.2 Side Window Options and Functions ... 755
29.17.3 Frame Layouts with Side Windows ... 757
29.18 Atomic Windows ... 759
29.19 Windows and Point ... 761
29.20 The Window Start and End Positions ... 762
29.21 Textual Scrolling ... 765
29.22 Vertical Fractional Scrolling ... 769
29.23 Horizontal Scrolling ... 770
29.24 Coordinates and Windows ... 772
29.25 Mouse Window Auto-selection ... 776
29.26 Window Configurations ... 777
29.27 Window Parameters ... 779
29.28 Hooks for Window Scrolling and Changes ... 782

30 Frames 787
30.1 Creating Frames 788
30.2 Multiple Terminals 789
30.3 Frame Geometry 793
30.3.1 Frame Layout 793
30.3.2 Frame Font 799
30.3.3 Frame Position 800
30.3.4 Frame Size 801
30.3.5 Implied Frame Resizing 803
30.4 Frame Parameters 804
30.4.1 Access to Frame Parameters 805
30.4.2 Initial Frame Parameters 806
30.4.3 Window Frame Parameters 807
30.4.3.1 Basic Parameters 807
30.4.3.2 Position Parameters 807
30.4.3.3 Size Parameters 810
30.4.3.4 Layout Parameters 812
30.4.3.5 Buffer Parameters 814
30.4.3.6 Frame Interaction Parameters 814
30.4.3.7 Mouse Dragging Parameters 816
30.4.3.8 Window Management Parameters 816
30.4.3.9 Cursor Parameters 819
30.4.3.10 Font and Color Parameters 820
30.4.4 Geometry 822
30.5 Terminal Parameters 823
30.6 Frame Titles 823
30.7 Deleting Frames 824
30.8 Finding All Frames 825
30.9 Minibuffers and Frames 826
30.10 Input Focus 827
30.11 Visibility of Frames 831
30.12 Raising, Lowering and Restacking Frames 832
30.13 Frame Configurations 833
30.14 Child Frames 833
30.15 Mouse Tracking 837
30.16 Mouse Position 838
30.17 Pop-Up Menus 839
30.18 Dialog Boxes 841
30.19 Pointer Shape 841
30.20 Window System Selections 842
30.21 Yanking Media 843
30.22 Drag and Drop 844
30.23 Color Names 848
30.24 Text Terminal Colors 850
30.25 X Resources 851
30.26 Display Feature Testing 852

31 Positions ... 855
31.1 Point ... 855
31.2 Motion ... 856
31.2.1 Motion by Characters ... 856
31.2.2 Motion by Words ... 857
31.2.3 Motion to an End of the Buffer ... 858
31.2.4 Motion by Text Lines ... 858
31.2.5 Motion by Screen Lines ... 860
31.2.6 Moving over Balanced Expressions ... 863
31.2.7 Skipping Characters ... 865
31.3 Excursions ... 866
31.4 Narrowing ... 867

32 Markers ... 871
32.1 Overview of Markers ... 871
32.2 Predicates on Markers ... 872
32.3 Functions that Create Markers ... 872
32.4 Information from Markers ... 874
32.5 Marker Insertion Types ... 874
32.6 Moving Marker Positions ... 875
32.7 The Mark ... 875
32.8 The Region ... 879

33 Text ... 880
33.1 Examining Text Near Point ... 880
33.2 Examining Buffer Contents ... 881
33.3 Comparing Text ... 884
33.4 Inserting Text ... 884
33.5 User-Level Insertion Commands ... 886
33.6 Deleting Text ... 888
33.7 User-Level Deletion Commands ... 889
33.8 The Kill Ring ... 891
33.8.1 Kill Ring Concepts ... 892
33.8.2 Functions for Killing ... 892
33.8.3 Yanking ... 893
33.8.4 Functions for Yanking ... 894
33.8.5 Low-Level Kill Ring ... 895
33.8.6 Internals of the Kill Ring ... 897
33.9 Undo ... 898
33.10 Maintaining Undo Lists ... 900
33.11 Filling ... 901
33.12 Margins for Filling ... 905
33.13 Adaptive Fill Mode ... 906
33.14 Auto Filling ... 907
33.15 Sorting Text ... 908

33.16 Counting Columns ... 912
33.17 Indentation ... 913
33.17.1 Indentation Primitives ... 913
33.17.2 Indentation Controlled by Major Mode ... 913
33.17.3 Indenting an Entire Region ... 915
33.17.4 Indentation Relative to Previous Lines ... 916
33.17.5 Adjustable Tab Stops ... 917
33.17.6 Indentation-Based Motion Commands ... 918
33.18 Case Changes ... 918
33.19 Text Properties ... 919
33.19.1 Examining Text Properties ... 920
33.19.2 Changing Text Properties ... 921
33.19.3 Text Property Search Functions ... 924
33.19.4 Properties with Special Meanings ... 927
33.19.5 Formatted Text Properties ... 934
33.19.6 Stickiness of Text Properties ... 935
33.19.7 Lazy Computation of Text Properties ... 936
33.19.8 Defining Clickable Text ... 937
33.19.9 Defining and Using Fields ... 939
33.19.10 Why Text Properties are not Intervals ... 941
33.20 Substituting for a Character Code ... 942
33.21 Registers ... 942
33.22 Transposition of Text ... 944
33.23 Replacing Buffer Text ... 944
33.24 Dealing With Compressed Data ... 945
33.25 Base 64 Encoding ... 946
33.26 Checksum/Hash ... 947
33.27 Suspicious Text ... 948
33.28 GnuTLS Cryptography ... 949
33.28.1 Format of GnuTLS Cryptography Inputs ... 950
33.28.2 GnuTLS Cryptographic Functions ... 950
33.29 Database ... 952
33.30 Parsing HTML and XML ... 955
33.30.1 Document Object Model ... 956
33.31 Parsing and generating JSON values ... 958
33.32 JSONRPC communication ... 959
33.32.1 Overview ... 960
33.32.2 Process-based JSONRPC connections ... 961
33.32.3 JSONRPC JSON object format ... 961
33.32.4 Deferred JSONRPC requests ... 962
33.33 Atomic Change Groups ... 962
33.34 Change Hooks ... 964

34 Non-ASCII Characters 966

34.1 Text Representations 966
34.2 Disabling Multibyte Characters 968
34.3 Converting Text Representations 968
34.4 Selecting a Representation 970
34.5 Character Codes 970
34.6 Character Properties 971
34.7 Character Sets 976
34.8 Scanning for Character Sets 978
34.9 Translation of Characters 978
34.10 Coding Systems 980
34.10.1 Basic Concepts of Coding Systems 980
34.10.2 Encoding and I/O 981
34.10.3 Coding Systems in Lisp 982
34.10.4 User-Chosen Coding Systems 985
34.10.5 Default Coding Systems 986
34.10.6 Specifying a Coding System for One Operation 989
34.10.7 Explicit Encoding and Decoding 991
34.10.8 Terminal I/O Encoding 993
34.11 Input Methods 994
34.12 Locales 995

35 Searching and Matching 996

35.1 Searching for Strings 996
35.2 Searching and Case 998
35.3 Regular Expressions 999
35.3.1 Syntax of Regular Expressions 999
35.3.1.1 Special Characters in Regular Expressions 999
35.3.1.2 Character Classes 1003
35.3.1.3 Backslash Constructs in Regular Expressions 1005
35.3.2 Complex Regexp Example 1008
35.3.3 Regular Expression Functions 1008
35.3.4 Problems with Regular Expressions 1010
35.4 Regular Expression Searching 1011
35.5 POSIX Regular Expression Searching 1014
35.6 The Match Data 1014
35.6.1 Replacing the Text that Matched 1015
35.6.2 Simple Match Data Access 1016
35.6.3 Accessing the Entire Match Data 1018
35.6.4 Saving and Restoring the Match Data 1019
35.7 Search and Replace 1019
35.8 Standard Regular Expressions Used in Editing 1023

36 Syntax Tables ... 1025
36.1 Syntax Table Concepts ... 1025
36.2 Syntax Descriptors ... 1026
36.2.1 Table of Syntax Classes ... 1026
36.2.2 Syntax Flags ... 1029
36.3 Syntax Table Functions ... 1030
36.4 Syntax Properties ... 1032
36.5 Motion and Syntax ... 1033
36.6 Parsing Expressions ... 1033
36.6.1 Motion Commands Based on Parsing ... 1034
36.6.2 Finding the Parse State for a Position ... 1035
36.6.3 Parser State ... 1035
36.6.4 Low-Level Parsing ... 1036
36.6.5 Parameters to Control Parsing ... 1037
36.7 Syntax Table Internals ... 1037
36.8 Categories ... 1039

37 Parsing Program Source ... 1042
37.1 Tree-sitter Language Grammar ... 1042
37.2 Using Tree-sitter Parser ... 1046
37.3 Retrieving Nodes ... 1048
37.4 Accessing Node Information ... 1054
37.5 Pattern Matching Tree-sitter Nodes ... 1055
37.6 Parsing Text in Multiple Languages ... 1060
37.7 Developing major modes with tree-sitter ... 1064
37.8 Tree-sitter C API Correspondence ... 1066

38 Abbrevs and Abbrev Expansion ... 1069
38.1 Abbrev Tables ... 1069
38.2 Defining Abbrevs ... 1070
38.3 Saving Abbrevs in Files ... 1071
38.4 Looking Up and Expanding Abbreviations ... 1072
38.5 Standard Abbrev Tables ... 1074
38.6 Abbrev Properties ... 1074
38.7 Abbrev Table Properties ... 1075

39 Threads ... 1076
39.1 Basic Thread Functions ... 1076
39.2 Mutexes ... 1078
39.3 Condition Variables ... 1078
39.4 The Thread List ... 1079

40 Processes 1081
40.1 Functions that Create Subprocesses 1081
40.2 Shell Arguments 1083
40.3 Creating a Synchronous Process 1084
40.4 Creating an Asynchronous Process 1089
40.5 Deleting Processes 1094
40.6 Process Information 1095
40.7 Sending Input to Processes 1099
40.8 Sending Signals to Processes 1100
40.9 Receiving Output from Processes 1102
40.9.1 Process Buffers 1102
40.9.2 Process Filter Functions 1104
40.9.3 Decoding Process Output 1106
40.9.4 Accepting Output from Processes 1107
40.9.5 Processes and Threads 1108
40.10 Sentinels: Detecting Process Status Changes 1109
40.11 Querying Before Exit 1110
40.12 Accessing Other Processes 1111
40.13 Transaction Queues 1113
40.14 Network Connections 1114
40.15 Network Servers 1117
40.16 Datagrams 1118
40.17 Low-Level Network Access 1118
40.17.1 `make-network-process` 1118
40.17.2 Network Options 1121
40.17.3 Testing Availability of Network Features 1122
40.18 Misc Network Facilities 1123
40.19 Communicating with Serial Ports 1125
40.20 Packing and Unpacking Byte Arrays 1128
40.20.1 Describing Data Layout 1128
40.20.2 Functions to Unpack and Pack Bytes 1130
40.20.3 Advanced data layout specifications 1131

41 Emacs Display 1133
41.1 Refreshing the Screen 1133
41.2 Forcing Redisplay 1133
41.3 Truncation 1134
41.4 The Echo Area 1135
41.4.1 Displaying Messages in the Echo Area 1135
41.4.2 Reporting Operation Progress 1138
41.4.3 Logging Messages in `*Messages*` 1141
41.4.4 Echo Area Customization 1142
41.5 Reporting Warnings 1142
41.5.1 Warning Basics 1143
41.5.2 Warning Variables 1144
41.5.3 Warning Options 1145

41.5.4 Delayed Warnings ... 1146
41.6 Invisible Text ... 1146
41.7 Selective Display ... 1149
41.8 Temporary Displays ... 1150
41.9 Overlays ... 1154
41.9.1 Managing Overlays ... 1154
41.9.2 Overlay Properties ... 1157
41.9.3 Searching for Overlays ... 1161
41.10 Size of Displayed Text ... 1161
41.11 Line Height ... 1166
41.12 Faces ... 1168
41.12.1 Face Attributes ... 1168
41.12.2 Defining Faces ... 1172
41.12.3 Face Attribute Functions ... 1175
41.12.4 Displaying Faces ... 1179
41.12.5 Face Remapping ... 1179
41.12.6 Functions for Working with Faces ... 1181
41.12.7 Automatic Face Assignment ... 1182
41.12.8 Basic Faces ... 1182
41.12.9 Font Selection ... 1184
41.12.10 Looking Up Fonts ... 1185
41.12.11 Fontsets ... 1186
41.12.12 Low-Level Font Representation ... 1188
41.13 Fringes ... 1193
41.13.1 Fringe Size and Position ... 1194
41.13.2 Fringe Indicators ... 1195
41.13.3 Fringe Cursors ... 1196
41.13.4 Fringe Bitmaps ... 1197
41.13.5 Customizing Fringe Bitmaps ... 1198
41.13.6 The Overlay Arrow ... 1199
41.14 Scroll Bars ... 1199
41.15 Window Dividers ... 1202
41.16 The `display` Property ... 1203
41.16.1 Display Specs That Replace The Text ... 1204
41.16.2 Specified Spaces ... 1205
41.16.3 Pixel Specification for Spaces ... 1206
41.16.4 Other Display Specifications ... 1207
41.16.5 Displaying in the Margins ... 1209
41.17 Images ... 1211
41.17.1 Image Formats ... 1211
41.17.2 Image Descriptors ... 1212
41.17.3 XBM Images ... 1216
41.17.4 XPM Images ... 1217
41.17.5 ImageMagick Images ... 1217
41.17.6 SVG Images ... 1218
41.17.7 Other Image Types ... 1224
41.17.8 Defining Images ... 1225

41.17.9 Showing Images ... 1227
41.17.10 Multi-Frame Images ... 1229
41.17.11 Image Cache ... 1230
41.18 Icons ... 1231
41.19 Embedded Native Widgets ... 1233
41.20 Buttons ... 1236
41.20.1 Button Properties ... 1237
41.20.2 Button Types ... 1238
41.20.3 Making Buttons ... 1238
41.20.4 Manipulating Buttons ... 1239
41.20.5 Button Buffer Commands ... 1240
41.21 Abstract Display ... 1241
41.21.1 Abstract Display Functions ... 1242
41.21.2 Abstract Display Example ... 1244
41.22 Blinking Parentheses ... 1246
41.23 Character Display ... 1246
41.23.1 Usual Display Conventions ... 1247
41.23.2 Display Tables ... 1248
41.23.3 Active Display Table ... 1249
41.23.4 Glyphs ... 1250
41.23.5 Glyphless Character Display ... 1250
41.24 Beeping ... 1252
41.25 Window Systems ... 1253
41.26 Tooltips ... 1254
41.27 Bidirectional Display ... 1255

42 Operating System Interface ... 1260

42.1 Starting Up Emacs ... 1260
42.1.1 Summary: Sequence of Actions at Startup ... 1260
42.1.2 The Init File ... 1263
42.1.3 Terminal-Specific Initialization ... 1265
42.1.4 Command-Line Arguments ... 1266
42.2 Getting Out of Emacs ... 1267
42.2.1 Killing Emacs ... 1267
42.2.2 Suspending Emacs ... 1268
42.3 Operating System Environment ... 1270
42.4 User Identification ... 1274
42.5 Time of Day ... 1275
42.6 Time Zone Rules ... 1277
42.7 Time Conversion ... 1278
42.8 Parsing and Formatting Times ... 1281
42.9 Processor Run time ... 1285
42.10 Time Calculations ... 1285
42.11 Timers for Delayed Execution ... 1286
42.12 Idle Timers ... 1289
42.13 Terminal Input ... 1290

42.13.1 Input Modes ... 1290
42.13.2 Recording Input ... 1291
42.14 Terminal Output ... 1292
42.15 Sound Output ... 1293
42.16 Operating on X11 Keysyms ... 1294
42.17 Batch Mode ... 1294
42.18 Session Management ... 1295
42.19 Desktop Notifications ... 1296
42.20 Notifications on File Changes ... 1301
42.21 Dynamically Loaded Libraries ... 1304
42.22 Security Considerations ... 1305

43 Preparing Lisp code for distribution ... 1308
43.1 Packaging Basics ... 1308
43.2 Simple Packages ... 1309
43.3 Multi-file Packages ... 1310
43.4 Creating and Maintaining Package Archives ... 1312
43.5 Interfacing to an archive web server ... 1313

Appendix A Emacs 28 Antinews ... 1314

Appendix B GNU Free Documentation License ... 1317

Appendix C GNU General Public License ... 1325

Appendix D Tips and Conventions ... 1336
D.1 Emacs Lisp Coding Conventions ... 1336
D.2 Key Binding Conventions ... 1339
D.3 Emacs Programming Tips ... 1340
D.4 Tips for Making Compiled Code Fast ... 1341
D.5 Tips for Avoiding Compiler Warnings ... 1342
D.6 Tips for Documentation Strings ... 1342
D.7 Tips on Writing Comments ... 1346
D.8 Conventional Headers for Emacs Libraries ... 1348

Appendix E GNU Emacs Internals 1351
E.1 Building Emacs 1351
E.2 Pure Storage 1354
E.3 Garbage Collection 1355
E.4 Stack-allocated Objects 1359
E.5 Memory Usage 1360
E.6 C Dialect 1360
E.7 Writing Emacs Primitives 1361
E.8 Writing Dynamically-Loaded Modules 1366
E.8.1 Module Initialization Code 1366
E.8.2 Writing Module Functions 1368
E.8.3 Conversion Between Lisp and Module Values 1371
E.8.4 Miscellaneous Convenience Functions for Modules 1378
E.8.5 Nonlocal Exits in Modules 1380
E.9 Object Internals 1382
E.9.1 Buffer Internals 1383
E.9.2 Window Internals 1387
E.9.3 Process Internals 1392
E.10 C Integer Types 1393

Appendix F Standard Errors 1395

Appendix G Standard Keymaps 1400

Appendix H Standard Hooks 1403

Index 1407

1 Introduction

Most of the GNU Emacs text editor is written in the programming language called Emacs Lisp. You can write new code in Emacs Lisp and install it as an extension to the editor. However, Emacs Lisp is more than a mere extension language; it is a full computer programming language in its own right. You can use it as you would any other programming language.

Because Emacs Lisp is designed for use in an editor, it has special features for scanning and parsing text as well as features for handling files, buffers, displays, subprocesses, and so on. Emacs Lisp is closely integrated with the editing facilities; thus, editing commands are functions that can also conveniently be called from Lisp programs, and parameters for customization are ordinary Lisp variables.

This manual attempts to be a full description of Emacs Lisp. For a beginner's introduction to Emacs Lisp, see *An Introduction to Emacs Lisp Programming*, by Bob Chassell, also published by the Free Software Foundation. This manual presumes considerable familiarity with the use of Emacs for editing; see *The GNU Emacs Manual* for this basic information.

Generally speaking, the earlier chapters describe features of Emacs Lisp that have counterparts in many programming languages, and later chapters describe features that are peculiar to Emacs Lisp or relate specifically to editing.

This is the *GNU Emacs Lisp Reference Manual*, corresponding to Emacs version 29.1.90.

1.1 Caveats

This manual has gone through numerous drafts. It is nearly complete but not flawless. There are a few topics that are not covered, either because we consider them secondary (such as most of the individual modes) or because they are yet to be written. Because we are not able to deal with them completely, we have left out several parts intentionally.

The manual should be fully correct in what it does cover, and it is therefore open to criticism on anything it says—from specific examples and descriptive text, to the ordering of chapters and sections. If something is confusing, or you find that you have to look at the sources or experiment to learn something not covered in the manual, then perhaps the manual should be fixed. Please let us know.

As you use this manual, we ask that you mark pages with corrections so you can later look them up and send them to us. If you think of a simple, real-life example for a function or group of functions, please make an effort to write it up and send it in. Please reference any comments to the chapter name, section name, and function name, as appropriate, since page numbers and chapter and section numbers will change and we may have trouble finding the text you are talking about. Also state the version of the edition you are criticizing.

Please send comments and corrections using `M-x report-emacs-bug`. For more details, See Section "Reporting Bugs" in *The GNU Emacs Manual*.

1.2 Lisp History

Lisp (LISt Processing language) was first developed in the late 1950s at the Massachusetts Institute of Technology for research in artificial intelligence. The great power of the Lisp language makes it ideal for other purposes as well, such as writing editing commands.

Dozens of Lisp implementations have been built over the years, each with its own idiosyncrasies. Many of them were inspired by Maclisp, which was written in the 1960s at MIT's Project MAC. Eventually the implementers of the descendants of Maclisp came together and developed a standard for Lisp systems, called Common Lisp. In the meantime, Gerry Sussman and Guy Steele at MIT developed a simplified but very powerful dialect of Lisp, called Scheme.

GNU Emacs Lisp is largely inspired by Maclisp, and a little by Common Lisp. If you know Common Lisp, you will notice many similarities. However, many features of Common Lisp have been omitted or simplified in order to reduce the memory requirements of GNU Emacs. Sometimes the simplifications are so drastic that a Common Lisp user might be very confused. We will occasionally point out how GNU Emacs Lisp differs from Common Lisp. If you don't know Common Lisp, don't worry about it; this manual is self-contained.

A certain amount of Common Lisp emulation is available via the `cl-lib` library. See Section "Overview" in *Common Lisp Extensions*.

Emacs Lisp is not at all influenced by Scheme; but the GNU project has an implementation of Scheme, called Guile. We use it in all new GNU software that calls for extensibility.

1.3 Conventions

This section explains the notational conventions that are used in this manual. You may want to skip this section and refer back to it later.

1.3.1 Some Terms

Throughout this manual, the phrases "the Lisp reader" and "the Lisp printer" refer to those routines in Lisp that convert textual representations of Lisp objects into actual Lisp objects, and vice versa. See Section 2.1 [Printed Representation], page 8, for more details. You, the person reading this manual, are thought of as the programmer and are addressed as "you". The user is the person who uses Lisp programs, including those you write.

Examples of Lisp code are formatted like this: `(list 1 2 3)`. Names that represent metasyntactic variables, or arguments to a function being described, are formatted like this: *first-number*.

1.3.2 `nil` and `t`

In Emacs Lisp, the symbol `nil` has three separate meanings: it is a symbol with the name '`nil`'; it is the logical truth value *false*; and it is the empty list—the list of zero elements. When used as a variable, `nil` always has the value `nil`.

As far as the Lisp reader is concerned, '`()`' and '`nil`' are identical: they stand for the same object, the symbol `nil`. The different ways of writing the symbol are intended entirely for human readers. After the Lisp reader has read either '`()`' or '`nil`', there is no way to determine which representation was actually written by the programmer.

In this manual, we write `()` when we wish to emphasize that it means the empty list, and we write `nil` when we wish to emphasize that it means the truth value *false*. That is a good convention to use in Lisp programs also.

```
(cons 'foo ())                ; Emphasize the empty list
(setq foo-flag nil)           ; Emphasize the truth value false
```

In contexts where a truth value is expected, any non-`nil` value is considered to be *true*. However, `t` is the preferred way to represent the truth value *true*. When you need to choose a value that represents *true*, and there is no other basis for choosing, use `t`. The symbol `t` always has the value `t`.

In Emacs Lisp, `nil` and `t` are special symbols that always evaluate to themselves. This is so that you do not need to quote them to use them as constants in a program. An attempt to change their values results in a `setting-constant` error. See Section 12.2 [Constant Variables], page 189.

`booleanp` *object* [Function]
Return non-`nil` if *object* is one of the two canonical boolean values: `t` or `nil`.

1.3.3 Evaluation Notation

A Lisp expression that you can evaluate is called a *form*. Evaluating a form always produces a result, which is a Lisp object. In the examples in this manual, this is indicated with '⇒':

```
(car '(1 2))
     ⇒ 1
```

You can read this as "`(car '(1 2))` evaluates to 1".

When a form is a macro call, it expands into a new form for Lisp to evaluate. We show the result of the expansion with '↦'. We may or may not show the result of the evaluation of the expanded form.

```
(third '(a b c))
     ↦ (car (cdr (cdr '(a b c))))
     ⇒ c
```

To help describe one form, we sometimes show another form that produces identical results. The exact equivalence of two forms is indicated with '≡'.

```
(make-sparse-keymap) ≡ (list 'keymap)
```

1.3.4 Printing Notation

Many of the examples in this manual print text when they are evaluated. If you execute example code in a Lisp Interaction buffer (such as the buffer `*scratch*`) by typing `C-j` after the closing parenthesis of the example, the printed text is inserted into the buffer. If you execute the example by other means (such as by evaluating the function `eval-region`), the printed text is displayed in the echo area.

Examples in this manual indicate printed text with '⊣', irrespective of where that text goes. The value returned by evaluating the form follows on a separate line with '⇒'.

```
(progn (prin1 'foo) (princ "\n") (prin1 'bar))
     ⊣ foo
     ⊣ bar
     ⇒ bar
```

1.3.5 Error Messages

Some examples signal errors. This normally displays an error message in the echo area. We show the error message on a line starting with 'error'. Note that 'error' itself does not appear in the echo area.

```
(+ 23 'x)
```

```
error  Wrong type argument: number-or-marker-p, x
```

1.3.6 Buffer Text Notation

Some examples describe modifications to the contents of a buffer, by showing the before and after versions of the text. These examples show the contents of the buffer in question between two lines of dashes containing the buffer name. In addition, '⋆' indicates the location of point. (The symbol for point, of course, is not part of the text in the buffer; it indicates the place *between* two characters where point is currently located.)

```
---------- Buffer: foo ----------
This is the ⋆contents of foo.
---------- Buffer: foo ----------

(insert "changed ")
     ⇒ nil
---------- Buffer: foo ----------
This is the changed ⋆contents of foo.
---------- Buffer: foo ----------
```

1.3.7 Format of Descriptions

Functions, variables, macros, commands, user options, and special forms are described in this manual in a uniform format. The first line of a description contains the name of the item followed by its arguments, if any. The category—function, variable, or whatever—is printed next to the right margin. The description follows on succeeding lines, sometimes with examples.

1.3.7.1 A Sample Function Description

In a function description, the name of the function being described appears first. It is followed on the same line by a list of argument names. These names are also used in the body of the description, to stand for the values of the arguments.

The appearance of the keyword `&optional` in the argument list indicates that the subsequent arguments may be omitted (omitted arguments default to `nil`). Do not write `&optional` when you call the function.

The keyword `&rest` (which must be followed by a single argument name) indicates that any number of arguments can follow. The single argument name following `&rest` receives, as its value, a list of all the remaining arguments passed to the function. Do not write `&rest` when you call the function.

Here is a description of an imaginary function `foo`:

`foo` *integer1* **&optional** *integer2* **&rest** *integers* [Function]

The function `foo` subtracts *integer1* from *integer2*, then adds all the rest of the arguments to the result. If *integer2* is not supplied, then the number 19 is used by default.

```
(foo 1 5 3 9)
     ⇒ 16
(foo 5)
     ⇒ 14
```

More generally,

```
(foo w x y...)
≡
(+ (- x w) y...)
```

By convention, any argument whose name contains the name of a type (e.g., *integer*, *integer1* or *buffer*) is expected to be of that type. A plural of a type (such as *buffers*) often means a list of objects of that type. An argument named *object* may be of any type. (For a list of Emacs object types, see Chapter 2 [Lisp Data Types], page 8.) An argument with any other sort of name (e.g., *new-file*) is specific to the function; if the function has a documentation string, the type of the argument should be described there (see Chapter 25 [Documentation], page 597).

See Section 13.2 [Lambda Expressions], page 232, for a more complete description of arguments modified by `&optional` and `&rest`.

Command, macro, and special form descriptions have the same format, but the word '`Function`' is replaced by '`Command`', '`Macro`', or '`Special Form`', respectively. Commands are simply functions that may be called interactively; macros process their arguments differently from functions (the arguments are not evaluated), but are presented the same way.

The descriptions of macros and special forms use a more complex notation to specify optional and repeated arguments, because they can break the argument list down into separate arguments in more complicated ways. '[*optional-arg*]' means that *optional-arg* is optional and '*repeated-args*...' stands for zero or more arguments. Parentheses are used when several arguments are grouped into additional levels of list structure. Here is an example:

`count-loop` (*var* [*from to* [*inc*]]) *body*... [Special Form]

This imaginary special form implements a loop that executes the *body* forms and then increments the variable *var* on each iteration. On the first iteration, the variable has the value *from*; on subsequent iterations, it is incremented by one (or by *inc* if that is given). The loop exits before executing *body* if *var* equals *to*. Here is an example:

```
(count-loop (i 0 10)
  (prin1 i) (princ " ")
  (prin1 (aref vector i))
  (terpri))
```

If *from* and *to* are omitted, *var* is bound to `nil` before the loop begins, and the loop exits if *var* is non-`nil` at the beginning of an iteration. Here is an example:

```
(count-loop (done)
  (if (pending)
      (fixit)
    (setq done t)))
```

In this special form, the arguments *from* and *to* are optional, but must both be present or both absent. If they are present, *inc* may optionally be specified as well. These arguments are grouped with the argument *var* into a list, to distinguish them from *body*, which includes all remaining elements of the form.

1.3.7.2 A Sample Variable Description

A *variable* is a name that can be *bound* (or *set*) to an object. The object to which a variable is bound is called a *value*; we say also that variable holds that value. Although nearly all variables can be set by the user, certain variables exist specifically so that users can change them; these are called *user options*. Ordinary variables and user options are described using a format like that for functions, except that there are no arguments.

Here is a description of the imaginary `electric-future-map` variable.

`electric-future-map` [Variable]
: The value of this variable is a full keymap used by Electric Command Future mode. The functions in this map allow you to edit commands you have not yet thought about executing.

User option descriptions have the same format, but '`Variable`' is replaced by '`User Option`'.

1.4 Version Information

These facilities provide information about which version of Emacs is in use.

`emacs-version` **&optional** *here* [Command]
: This function returns a string describing the version of Emacs that is running. It is useful to include this string in bug reports.

```
(emacs-version)
  ⇒ "GNU Emacs 26.1 (build 1, x86_64-unknown-linux-gnu,
             GTK+ Version 3.16) of 2017-06-01"
```

: If *here* is non-`nil`, it inserts the text in the buffer before point, and returns `nil`. When this function is called interactively, it prints the same information in the echo area, but giving a prefix argument makes *here* non-`nil`.

`emacs-build-time` [Variable]
: The value of this variable indicates the time at which Emacs was built. It uses the style of `current-time` (see Section 42.5 [Time of Day], page 1275), or is `nil` if the information is not available.

```
emacs-build-time
     ⇒ (25194 55894 8547 617000)
```

: (This timestamp is `(1651169878008547617 . 1000000000)` if `current-time-list` was `nil` when Emacs was built.)

`emacs-version` [Variable]
: The value of this variable is the version of Emacs being run. It is a string such as `"26.1"`. A value with three numeric components, such as `"26.0.91"`, indicates an unreleased test version. (Prior to Emacs 26.1, the string includes an extra final component with the integer that is now stored in `emacs-build-number`; e.g., `"25.1.1"`.)

`emacs-major-version` [Variable]
: The major version number of Emacs, as an integer. For Emacs version 23.1, the value is 23.

`emacs-minor-version` [Variable]
: The minor version number of Emacs, as an integer. For Emacs version 23.1, the value is 1.

`emacs-build-number` [Variable]
: An integer that increments each time Emacs is built in the same directory (without cleaning). This is only of relevance when developing Emacs.

`emacs-repository-version` [Variable]
: A string that gives the repository revision from which Emacs was built. If Emacs was built outside revision control, the value is `nil`.

`emacs-repository-branch` [Variable]
: A string that gives the repository branch from which Emacs was built. In the most cases this is `"master"`. If Emacs was built outside revision control, the value is `nil`.

1.5 Acknowledgments

This manual was originally written by Robert Krawitz, Bil Lewis, Dan LaLiberte, Richard M. Stallman and Chris Welty, the volunteers of the GNU manual group, in an effort extending over several years. Robert J. Chassell helped to review and edit the manual, with the support of the Defense Advanced Research Projects Agency, ARPA Order 6082, arranged by Warren A. Hunt, Jr. of Computational Logic, Inc. Additional sections have since been written by Miles Bader, Lars Brinkhoff, Chong Yidong, Kenichi Handa, Lute Kamstra, Juri Linkov, Glenn Morris, Thien-Thi Nguyen, Dan Nicolaescu, Martin Rudalics, Kim F. Storm, Luc Teirlinck, and Eli Zaretskii, and others.

Corrections were supplied by Drew Adams, Juanma Barranquero, Karl Berry, Jim Blandy, Bard Bloom, Stephane Boucher, David Boyes, Alan Carroll, Richard Davis, Lawrence R. Dodd, Peter Doornbosch, David A. Duff, Chris Eich, Beverly Erlebacher, David Eckelkamp, Ralf Fassel, Eirik Fuller, Stephen Gildea, Bob Glickstein, Eric Hanchrow, Jesper Harder, George Hartzell, Nathan Hess, Masayuki Ida, Dan Jacobson, Jak Kirman, Bob Knighten, Frederick M. Korz, Joe Lammens, Glenn M. Lewis, K. Richard Magill, Brian Marick, Roland McGrath, Stefan Monnier, Skip Montanaro, John Gardiner Myers, Thomas A. Peterson, Francesco Potortì, Friedrich Pukelsheim, Arnold D. Robbins, Raul Rockwell, Jason Rumney, Per Starbäck, Shinichirou Sugou, Kimmo Suominen, Edward Tharp, Bill Trost, Rickard Westman, Jean White, Eduard Wiebe, Matthew Wilding, Carl Witty, Dale Worley, Rusty Wright, and David D. Zuhn.

For a more complete list of contributors, please see the relevant change log entries in the Emacs source repository.

2 Lisp Data Types

A Lisp *object* is a piece of data used and manipulated by Lisp programs. For our purposes, a *type* or *data type* is a set of possible objects.

Every object belongs to at least one type. Objects of the same type have similar structures and may usually be used in the same contexts. Types can overlap, and objects can belong to two or more types. Consequently, we can ask whether an object belongs to a particular type, but not for *the* type of an object.

A few fundamental object types are built into Emacs. These, from which all other types are constructed, are called *primitive types*. Each object belongs to one and only one primitive type. These types include *integer*, *float*, *cons*, *symbol*, *string*, *vector*, *hash-table*, *subr*, *byte-code function*, and *record*, plus several special types, such as *buffer*, that are related to editing. (See Section 2.5 [Editing Types], page 26.)

Each primitive type has a corresponding Lisp function that checks whether an object is a member of that type.

Lisp is unlike many other languages in that its objects are *self-typing*: the primitive type of each object is implicit in the object itself. For example, if an object is a vector, nothing can treat it as a number; Lisp knows it is a vector, not a number.

In most languages, the programmer must declare the data type of each variable, and the type is known by the compiler but not represented in the data. Such type declarations do not exist in Emacs Lisp. A Lisp variable can have any type of value, and it remembers whatever value you store in it, type and all. (Actually, a small number of Emacs Lisp variables can only take on values of a certain type. See Section 12.16 [Variables with Restricted Values], page 223.)

This chapter describes the purpose, printed representation, and read syntax of each of the standard types in GNU Emacs Lisp. Details on how to use these types can be found in later chapters.

2.1 Printed Representation and Read Syntax

The *printed representation* of an object is the format of the output generated by the Lisp printer (the function `prin1`) for that object. Every data type has a unique printed representation. The *read syntax* of an object is the format of the input accepted by the Lisp reader (the function `read`) for that object. This is not necessarily unique; many kinds of object have more than one syntax. See Chapter 20 [Read and Print], page 372.

In most cases, an object's printed representation is also a read syntax for the object. However, some types have no read syntax, since it does not make sense to enter objects of these types as constants in a Lisp program. These objects are printed in *hash notation*, which consists of the characters '`#<`', a descriptive string (typically the type name followed by the name of the object), and a closing '`>`'. (This is called "hash notation" because it begins with the '`#`' character, known as "hash" or "number sign"). For example:

```
(current-buffer)
     ⇒ #<buffer objects.texi>
```

Hash notation cannot be read at all, so the Lisp reader signals the error `invalid-read-syntax` whenever it encounters '`#<`'.

We describe the read syntax and the printed representation of each Lisp data type where we describe that data type, in the following sections of this chapter. For example, see Section 2.4.8 [String Type], page 20, and its subsections for the read syntax and printed representation of strings; see Section 2.4.9 [Vector Type], page 22, for the same information about vectors; etc.

In other languages, an expression is text; it has no other form. In Lisp, an expression is primarily a Lisp object and only secondarily the text that is the object's read syntax. Often there is no need to emphasize this distinction, but you must keep it in the back of your mind, or you will occasionally be very confused.

When you evaluate an expression interactively, the Lisp interpreter first reads the textual representation of it, producing a Lisp object, and then evaluates that object (see Chapter 10 [Evaluation], page 145). However, evaluation and reading are separate activities. Reading returns the Lisp object represented by the text that is read; the object may or may not be evaluated later. See Section 20.3 [Input Functions], page 375, for a description of `read`, the basic function for reading objects.

2.2 Special Read Syntax

Emacs Lisp represents many special objects and constructs via special hash notations.

'`#<...>`' Objects that have no read syntax are presented like this (see Section 2.1 [Printed Representation], page 8).

'`##`' The printed representation of an interned symbol whose name is an empty string (see Section 2.4.4 [Symbol Type], page 14).

'`#'`' This is a shortcut for `function`, see Section 13.7 [Anonymous Functions], page 243.

'`#:`' The printed representation of an uninterned symbol whose name is *foo* is '`#:foo`' (see Section 2.4.4 [Symbol Type], page 14).

'`#N`' When printing circular structures, this construct is used to represent where the structure loops back onto itself, and '`N`' is the starting list count:

```
(let ((a (list 1)))
  (setcdr a a))
=> (1 . #0)
```

'`#N=`'

'`#N#`' '`#N=`' gives the name to an object, and '`#N#`' represents that object, so when reading back the object, they will be the same object instead of copies (see Section 2.6 [Circular Objects], page 30).

'`#xN`' '`N`' represented as a hexadecimal number ('`#x2a`').

'`#oN`' '`N`' represented as an octal number ('`#o52`').

'`#bN`' '`N`' represented as a binary number ('`#b101010`').

'`#(...)`' String text properties (see Section 2.4.8.4 [Text Props and Strings], page 21).

'`#^`' A char table (see Section 2.4.10 [Char-Table Type], page 22).

‘`#s(hash-table ...)`’
: A hash table (see Section 2.4.12 [Hash Table Type], page 23).

‘`?C`’
: A character (see Section 2.4.3.1 [Basic Char Syntax], page 11).

‘`#$`’
: The current file name in byte-compiled files (see Section 17.3 [Docs and Compilation], page 320). This is not meant to be used in Emacs Lisp source files.

‘`#@N`’
: Skip the next ‘`N`’ characters (see Section 2.3 [Comments], page 10). This is used in byte-compiled files, and is not meant to be used in Emacs Lisp source files.

‘`#f`’
: Indicates that the following form isn't readable by the Emacs Lisp reader. This is only in text for display purposes (when that would look prettier than alternative ways of indicating an unreadable form) and will never appear in any Lisp file.

2.3 Comments

A *comment* is text that is written in a program only for the sake of humans that read the program, and that has no effect on the meaning of the program. In Lisp, an unescaped semicolon (‘`;`’) starts a comment if it is not within a string or character constant. The comment continues to the end of line. The Lisp reader discards comments; they do not become part of the Lisp objects which represent the program within the Lisp system.

The ‘`#@`*count*’ construct, which skips the next *count* characters, is useful for program-generated comments containing binary data. The Emacs Lisp byte compiler uses this in its output files (see Chapter 17 [Byte Compilation], page 317). It isn't meant for source files, however.

See Section D.7 [Comment Tips], page 1346, for conventions for formatting comments.

2.4 Programming Types

There are two general categories of types in Emacs Lisp: those having to do with Lisp programming, and those having to do with editing. The former exist in many Lisp implementations, in one form or another. The latter are unique to Emacs Lisp.

2.4.1 Integer Type

Under the hood, there are two kinds of integers—small integers, called *fixnums*, and large integers, called *bignums*.

The range of values for a fixnum depends on the machine. The minimum range is −536,870,912 to 536,870,911 (30 bits; i.e., -2^{29} to $2^{29} - 1$) but many machines provide a wider range.

Bignums can have arbitrary precision. Operations that overflow a fixnum will return a bignum instead.

All numbers can be compared with `eql` or `=`; fixnums can also be compared with `eq`. To test whether an integer is a fixnum or a bignum, you can compare it to `most-negative-fixnum` and `most-positive-fixnum`, or you can use the convenience predicates `fixnump` and `bignump` on any object.

The read syntax for integers is a sequence of (base ten) digits with an optional sign at the beginning and an optional period at the end. The printed representation produced by the Lisp interpreter never has a leading '`+`' or a final '`.`'.

```
-1              ; The integer −1.
1               ; The integer 1.
1.              ; Also the integer 1.
+1              ; Also the integer 1.
```

See Chapter 3 [Numbers], page 38, for more information.

2.4.2 Floating-Point Type

Floating-point numbers are the computer equivalent of scientific notation; you can think of a floating-point number as a fraction together with a power of ten. The precise number of significant figures and the range of possible exponents is machine-specific; Emacs uses the C data type `double` to store the value, and internally this records a power of 2 rather than a power of 10.

The printed representation for floating-point numbers requires either a decimal point (with at least one digit following), an exponent, or both. For example, '`1500.0`', '`+15e2`', '`15.0e+2`', '`+1500000e-3`', and '`.15e4`' are five ways of writing a floating-point number whose value is 1500. They are all equivalent.

See Chapter 3 [Numbers], page 38, for more information.

2.4.3 Character Type

A *character* in Emacs Lisp is nothing more than an integer. In other words, characters are represented by their character codes. For example, the character `A` is represented as the integer 65. That is also their usual printed representation; see Section 2.4.3.1 [Basic Char Syntax], page 11.

Individual characters are used occasionally in programs, but it is more common to work with *strings*, which are sequences composed of characters. See Section 2.4.8 [String Type], page 20.

Characters in strings and buffers are currently limited to the range of 0 to 4194303—twenty two bits (see Section 34.5 [Character Codes], page 970). Codes 0 through 127 are ASCII codes; the rest are non-ASCII (see Chapter 34 [Non-ASCII Characters], page 966). Characters that represent keyboard input have a much wider range, to encode modifier keys such as Control, Meta and Shift.

There are special functions for producing a human-readable textual description of a character for the sake of messages. See Section 25.5 [Describing Characters], page 603.

2.4.3.1 Basic Char Syntax

Since characters are really integers, the printed representation of a character is a decimal number. This is also a possible read syntax for a character, but writing characters that way in Lisp programs is not clear programming. You should *always* use the special read syntax formats that Emacs Lisp provides for characters. These syntax formats start with a question mark.

The usual read syntax for alphanumeric characters is a question mark followed by the character; thus, '?A' for the character A, '?B' for the character B, and '?a' for the character a.

For example:

```
?Q ⇒ 81     ?q ⇒ 113
```

You can use the same syntax for punctuation characters. However, if the punctuation character has a special syntactic meaning in Lisp, you must quote it with a '\'. For example, '?\(' is the way to write the open-paren character. Likewise, if the character is '\', you must use a second '\' to quote it: '?\\'.

You can express the characters control-g, backspace, tab, newline, vertical tab, formfeed, space, return, del, and escape as '?\a', '?\b', '?\t', '?\n', '?\v', '?\f', '?\s', '?\r', '?\d', and '?\e', respectively. ('?\s' followed by a dash has a different meaning—it applies the Super modifier to the following character.) Thus,

```
?\a ⇒ 7                 ; control-g, C-g
?\b ⇒ 8                 ; backspace, BS, C-h
?\t ⇒ 9                 ; tab, TAB, C-i
?\n ⇒ 10                ; newline, C-j
?\v ⇒ 11                ; vertical tab, C-k
?\f ⇒ 12                ; formfeed character, C-l
?\r ⇒ 13                ; carriage return, RET, C-m
?\e ⇒ 27                ; escape character, ESC, C-[
?\s ⇒ 32                ; space character, SPC
?\\ ⇒ 92                ; backslash character, \
?\d ⇒ 127               ; delete character, DEL
```

These sequences which start with backslash are also known as *escape sequences*, because backslash plays the role of an escape character; this has nothing to do with the character ESC. '\s' is meant for use in character constants; in string constants, just write the space.

A backslash is allowed, and harmless, preceding any character without a special escape meaning; thus, '?\+' is equivalent to '?+'. There is no reason to add a backslash before most characters. However, you must add a backslash before any of the characters '()[]\;"', and you should add a backslash before any of the characters '|'`#.,' to avoid confusing the Emacs commands for editing Lisp code. You should also add a backslash before Unicode characters which resemble the previously mentioned ASCII ones, to avoid confusing people reading your code. Emacs will highlight some non-escaped commonly confused characters such as '‘' to encourage this. You can also add a backslash before whitespace characters such as space, tab, newline and formfeed. However, it is cleaner to use one of the easily readable escape sequences, such as '\t' or '\s', instead of an actual whitespace character such as a tab or a space. (If you do write backslash followed by a space, you should write an extra space after the character constant to separate it from the following text.)

2.4.3.2 General Escape Syntax

In addition to the specific escape sequences for special important control characters, Emacs provides several types of escape syntax that you can use to specify non-ASCII text characters.

1. You can specify characters by their Unicode names, if any. ?\N{*NAME*} represents the Unicode character named *NAME*. Thus, '?\N{LATIN SMALL LETTER A WITH GRAVE}' is

equivalent to `?à` and denotes the Unicode character U+00E0. To simplify entering multi-line strings, you can replace spaces in the names by non-empty sequences of whitespace (e.g., newlines).

2. You can specify characters by their Unicode values. `?\N{U+X}` represents a character with Unicode code point *X*, where *X* is a hexadecimal number. Also, `?\uxxxx` and `?\Uxxxxxxxx` represent code points *xxxx* and *xxxxxxxx*, respectively, where each *x* is a single hexadecimal digit. For example, `?\N{U+E0}`, `?\u00e0` and `?\U000000E0` are all equivalent to `?à` and to '`?\N{LATIN SMALL LETTER A WITH GRAVE}`'. The Unicode Standard defines code points only up to '`U+10ffff`', so if you specify a code point higher than that, Emacs signals an error.
3. You can specify characters by their hexadecimal character codes. A hexadecimal escape sequence consists of a backslash, '`x`', and the hexadecimal character code. Thus, '`?\x41`' is the character `A`, '`?\x1`' is the character `C-a`, and `?\xe0` is the character `à` (`a` with grave accent). You can use one or more hex digits after '`x`', so you can represent any character code in this way.
4. You can specify characters by their character code in octal. An octal escape sequence consists of a backslash followed by up to three octal digits; thus, '`?\101`' for the character `A`, '`?\001`' for the character `C-a`, and `?\002` for the character `C-b`. Only characters up to octal code 777 can be specified this way.

These escape sequences may also be used in strings. See Section 2.4.8.2 [Non-ASCII in Strings], page 20.

2.4.3.3 Control-Character Syntax

Control characters can be represented using yet another read syntax. This consists of a question mark followed by a backslash, caret, and the corresponding non-control character, in either upper or lower case. For example, both '`?\^I`' and '`?\^i`' are valid read syntax for the character `C-i`, the character whose value is 9.

Instead of the '`^`', you can use '`C-`'; thus, '`?\C-i`' is equivalent to '`?\^I`' and to '`?\^i`':

```
?\^I ⇒ 9     ?\C-I ⇒ 9
```

In strings and buffers, the only control characters allowed are those that exist in ASCII; but for keyboard input purposes, you can turn any character into a control character with '`C-`'. The character codes for these non-ASCII control characters include the 2^{26} bit as well as the code for the corresponding non-control character. Not all text terminals can generate non-ASCII control characters, but it is straightforward to generate them using X and other window systems.

For historical reasons, Emacs treats the `DEL` character as the control equivalent of `?`:

```
?\^? ⇒ 127     ?\C-? ⇒ 127
```

As a result, it is currently not possible to represent the character `Control-?`, which is a meaningful input character under X, using '`\C-`'. It is not easy to change this, as various Lisp files refer to `DEL` in this way.

For representing control characters to be found in files or strings, we recommend the '`^`' syntax; for control characters in keyboard input, we prefer the '`C-`' syntax. Which one you use does not affect the meaning of the program, but may guide the understanding of people who read it.

2.4.3.4 Meta-Character Syntax

A *meta character* is a character typed with the `META` modifier key. The integer that represents such a character has the 2^{27} bit set. We use high bits for this and other modifiers to make possible a wide range of basic character codes.

In a string, the 2^7 bit attached to an ASCII character indicates a meta character; thus, the meta characters that can fit in a string have codes in the range from 128 to 255, and are the meta versions of the ordinary ASCII characters. See Section 22.7.17 [Strings of Events], page 460, for details about `META`-handling in strings.

The read syntax for meta characters uses '`\M-`'. For example, '`?\M-A`' stands for *M-A*. You can use '`\M-`' together with octal character codes (see below), with '`\C-`', or with any other syntax for a character. Thus, you can write *M-A* as '`?\M-A`', or as '`?\M-\101`'. Likewise, you can write *C-M-b* as '`?\M-\C-b`', '`?\C-\M-b`', or '`?\M-\002`'.

2.4.3.5 Other Character Modifier Bits

The case of a graphic character is indicated by its character code; for example, ASCII distinguishes between the characters '`a`' and '`A`'. But ASCII has no way to represent whether a control character is upper case or lower case. Emacs uses the 2^{25} bit to indicate that the shift key was used in typing a control character. This distinction is possible only on a graphical display such as a GUI display on X; text terminals do not report the distinction. The Lisp syntax for the shift bit is '`\S-`'; thus, '`?\C-\S-o`' or '`?\C-\S-O`' represents the shifted-control-o character.

The X Window System defines three other modifier bits that can be set in a character: *hyper*, *super* and *alt*. The syntaxes for these bits are '`\H-`', '`\s-`' and '`\A-`'. (Case is significant in these prefixes.) Thus, '`?\H-\M-\A-x`' represents *Alt-Hyper-Meta-x*. (Note that '`\s`' with no following '`-`' represents the space character.) Numerically, the bit values are 2^{22} for alt, 2^{23} for super and 2^{24} for hyper.

2.4.4 Symbol Type

A *symbol* in GNU Emacs Lisp is an object with a name. The symbol name serves as the printed representation of the symbol. In ordinary Lisp use, with one single obarray (see Section 9.3 [Creating Symbols], page 135), a symbol's name is unique—no two symbols have the same name.

A symbol can serve as a variable, as a function name, or to hold a property list. Or it may serve only to be distinct from all other Lisp objects, so that its presence in a data structure may be recognized reliably. In a given context, usually only one of these uses is intended. But you can use one symbol in all of these ways, independently.

A symbol whose name starts with a colon ('`:`') is called a *keyword symbol*. These symbols automatically act as constants, and are normally used only by comparing an unknown symbol with a few specific alternatives. See Section 12.2 [Constant Variables], page 189.

A symbol name can contain any characters whatever. Most symbol names are written with letters, digits, and the punctuation characters '`-+=*/`'. Such names require no special punctuation; the characters of the name suffice as long as the name does not look like a number. (If it does, write a '`\`' at the beginning of the name to force interpretation as a symbol.) The characters '`_~!@$%^&:<>{}?`' are less often used but also require no special punctuation. Any other characters may be included in a symbol's name by escaping them

with a backslash. In contrast to its use in strings, however, a backslash in the name of a symbol simply quotes the single character that follows the backslash. For example, in a string, '`\t`' represents a tab character; in the name of a symbol, however, '`\t`' merely quotes the letter '`t`'. To have a symbol with a tab character in its name, you must actually use a tab (preceded with a backslash). But it's rare to do such a thing.

> **Common Lisp note:** In Common Lisp, lower case letters are always folded to upper case, unless they are explicitly escaped. In Emacs Lisp, upper case and lower case letters are distinct.

Here are several examples of symbol names. Note that the '`+`' in the fourth example is escaped to prevent it from being read as a number. This is not necessary in the sixth example because the rest of the name makes it invalid as a number.

```
foo                 ; A symbol named 'foo'.
FOO                 ; A symbol named 'FOO', different from 'foo'.
1+                  ; A symbol named '1+'
                    ;   (not '+1', which is an integer).
\+1                 ; A symbol named '+1'
                    ;   (not a very readable name).
\(*\ 1\ 2\)         ; A symbol named '(* 1 2)' (a worse name).
+-*/_~!@$%^&=:<>{}  ; A symbol named '+-*/_~!@$%^&=:<>{}'.
                    ;   These characters need not be escaped.
```

As an exception to the rule that a symbol's name serves as its printed representation, '`##`' is the printed representation for an interned symbol whose name is an empty string. Furthermore, '`#:`*foo*' is the printed representation for an uninterned symbol whose name is *foo*. (Normally, the Lisp reader interns all symbols; see Section 9.3 [Creating Symbols], page 135.)

2.4.5 Sequence Types

A *sequence* is a Lisp object that represents an ordered set of elements. There are two kinds of sequence in Emacs Lisp: *lists* and *arrays*.

Lists are the most commonly-used sequences. A list can hold elements of any type, and its length can be easily changed by adding or removing elements. See the next subsection for more about lists.

Arrays are fixed-length sequences. They are further subdivided into strings, vectors, char-tables and bool-vectors. Vectors can hold elements of any type, whereas string elements must be characters, and bool-vector elements must be `t` or `nil`. Char-tables are like vectors except that they are indexed by any valid character code. The characters in a string can have text properties like characters in a buffer (see Section 33.19 [Text Properties], page 919), but vectors do not support text properties, even when their elements happen to be characters.

Lists, strings and the other array types also share important similarities. For example, all have a length *l*, and all have elements which can be indexed from zero to *l* minus one. Several functions, called sequence functions, accept any kind of sequence. For example, the function `length` reports the length of any kind of sequence. See Chapter 6 [Sequences Arrays Vectors], page 102.

It is generally impossible to read the same sequence twice, since sequences are always created anew upon reading. If you read the read syntax for a sequence twice, you get two

sequences with equal contents. There is one exception: the empty list `()` always stands for the same object, `nil`.

2.4.6 Cons Cell and List Types

A *cons cell* is an object that consists of two slots, called the CAR slot and the CDR slot. Each slot can *hold* any Lisp object. We also say that the CAR of this cons cell is whatever object its CAR slot currently holds, and likewise for the CDR.

A *list* is a series of cons cells, linked together so that the CDR slot of each cons cell holds either the next cons cell or the empty list. The empty list is actually the symbol `nil`. See Chapter 5 [Lists], page 77, for details. Because most cons cells are used as part of lists, we refer to any structure made out of cons cells as a *list structure*.

> A note to C programmers: a Lisp list thus works as a *linked list* built up of cons cells. Because pointers in Lisp are implicit, we do not distinguish between a cons cell slot holding a value versus pointing to the value.

Because cons cells are so central to Lisp, we also have a word for an object which is not a cons cell. These objects are called *atoms*.

The read syntax and printed representation for lists are identical, and consist of a left parenthesis, an arbitrary number of elements, and a right parenthesis. Here are examples of lists:

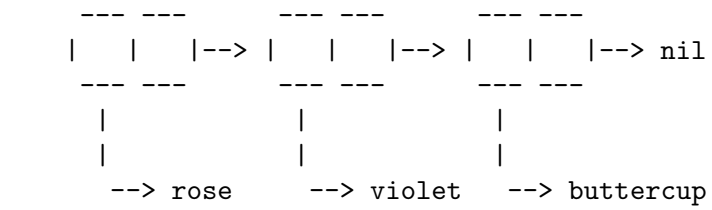
```
(A 2 "A")           ; A list of three elements.
()                  ; A list of no elements (the empty list).
nil                 ; A list of no elements (the empty list).
("A ()")            ; A list of one element: the string "A ()".
(A ())              ; A list of two elements: A and the empty list.
(A nil)             ; Equivalent to the previous.
((A B C))           ; A list of one element
                    ;   (which is a list of three elements).
```

Upon reading, each object inside the parentheses becomes an element of the list. That is, a cons cell is made for each element. The CAR slot of the cons cell holds the element, and its CDR slot refers to the next cons cell of the list, which holds the next element in the list. The CDR slot of the last cons cell is set to hold `nil`.

The names CAR and CDR derive from the history of Lisp. The original Lisp implementation ran on an IBM 704 computer which divided words into two parts, the address and the decrement; CAR was an instruction to extract the contents of the address part of a register, and CDR an instruction to extract the contents of the decrement. By contrast, cons cells are named for the function `cons` that creates them, which in turn was named for its purpose, the construction of cells.

2.4.6.1 Drawing Lists as Box Diagrams

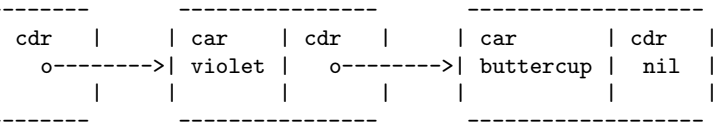
A list can be illustrated by a diagram in which the cons cells are shown as pairs of boxes, like dominoes. (The Lisp reader cannot read such an illustration; unlike the textual notation, which can be understood by both humans and computers, the box illustrations can be understood only by humans.) This picture represents the three-element list `(rose violet buttercup)`:



In this diagram, each box represents a slot that can hold or refer to any Lisp object. Each pair of boxes represents a cons cell. Each arrow represents a reference to a Lisp object, either an atom or another cons cell.

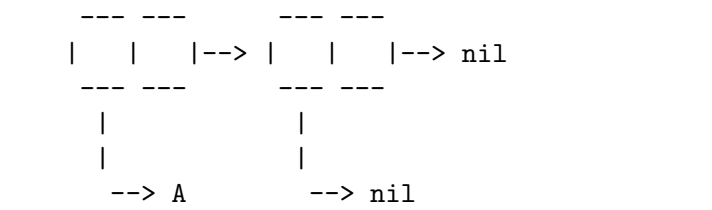
In this example, the first box, which holds the CAR of the first cons cell, refers to or holds `rose` (a symbol). The second box, holding the CDR of the first cons cell, refers to the next pair of boxes, the second cons cell. The CAR of the second cons cell is `violet`, and its CDR is the third cons cell. The CDR of the third (and last) cons cell is `nil`.

Here is another diagram of the same list, `(rose violet buttercup)`, sketched in a different manner:

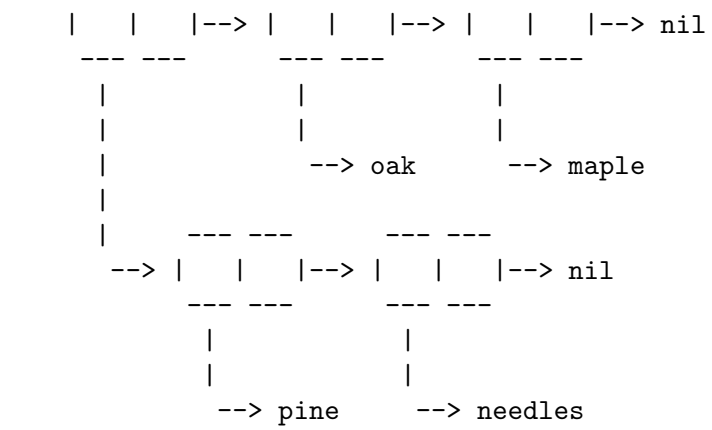


A list with no elements in it is the *empty list*; it is identical to the symbol `nil`. In other words, `nil` is both a symbol and a list.

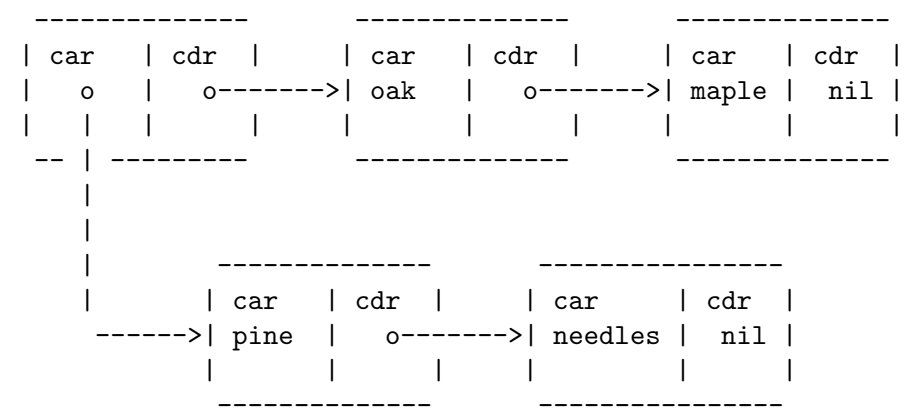
Here is the list `(A ())`, or equivalently `(A nil)`, depicted with boxes and arrows:



Here is a more complex illustration, showing the three-element list, `((pine needles) oak maple)`, the first element of which is a two-element list:



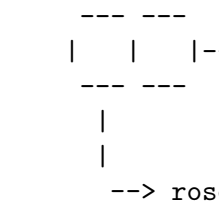
The same list represented in the second box notation looks like this:



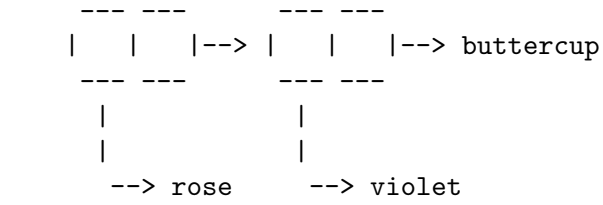
2.4.6.2 Dotted Pair Notation

Dotted pair notation is a general syntax for cons cells that represents the CAR and CDR explicitly. In this syntax, `(a . b)` stands for a cons cell whose CAR is the object *a* and whose CDR is the object *b*. Dotted pair notation is more general than list syntax because the CDR does not have to be a list. However, it is more cumbersome in cases where list syntax would work. In dotted pair notation, the list '`(1 2 3)`' is written as '`(1 . (2 . (3 . nil)))`'. For `nil`-terminated lists, you can use either notation, but list notation is usually clearer and more convenient. When printing a list, the dotted pair notation is only used if the CDR of a cons cell is not a list.

Here's an example using boxes to illustrate dotted pair notation. This example shows the pair `(rose . violet)`:

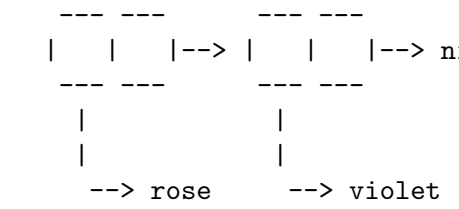


You can combine dotted pair notation with list notation to represent conveniently a chain of cons cells with a non-`nil` final CDR. You write a dot after the last element of the list, followed by the CDR of the final cons cell. For example, `(rose violet . buttercup)` is equivalent to `(rose . (violet . buttercup))`. The object looks like this:



The syntax `(rose . violet . buttercup)` is invalid because there is nothing that it could mean. If anything, it would say to put `buttercup` in the CDR of a cons cell whose CDR is already used for `violet`.

The list `(rose violet)` is equivalent to `(rose . (violet))`, and looks like this:



Similarly, the three-element list `(rose violet buttercup)` is equivalent to `(rose . (violet . (buttercup)))`.

2.4.6.3 Association List Type

An *association list* or *alist* is a specially-constructed list whose elements are cons cells. In each element, the CAR is considered a *key*, and the CDR is considered an *associated value*. (In some cases, the associated value is stored in the CAR of the CDR.) Association lists are often used as stacks, since it is easy to add or remove associations at the front of the list.

For example,

```
(setq alist-of-colors
      '((rose . red) (lily . white) (buttercup . yellow)))
```

sets the variable `alist-of-colors` to an alist of three elements. In the first element, `rose` is the key and `red` is the value.

See Section 5.8 [Association Lists], page 95, for a further explanation of alists and for functions that work on alists. See Chapter 8 [Hash Tables], page 127, for another kind of lookup table, which is much faster for handling a large number of keys.

2.4.7 Array Type

An *array* is composed of an arbitrary number of slots for holding or referring to other Lisp objects, arranged in a contiguous block of memory. Accessing any element of an array takes approximately the same amount of time. In contrast, accessing an element of a list requires time proportional to the position of the element in the list. (Elements at the end of a list take longer to access than elements at the beginning of a list.)

Emacs defines four types of array: strings, vectors, bool-vectors, and char-tables.

A string is an array of characters and a vector is an array of arbitrary objects. A bool-vector can hold only `t` or `nil`. These kinds of array may have any length up to the largest fixnum, subject to system architecture limits and available memory. Char-tables are sparse arrays indexed by any valid character code; they can hold arbitrary objects.

The first element of an array has index zero, the second element has index 1, and so on. This is called *zero-origin* indexing. For example, an array of four elements has indices 0, 1, 2, and 3. The largest possible index value is one less than the length of the array. Once an array is created, its length is fixed.

All Emacs Lisp arrays are one-dimensional. (Most other programming languages support multidimensional arrays, but they are not essential; you can get the same effect with nested one-dimensional arrays.) Each type of array has its own read syntax; see the following sections for details.

The array type is a subset of the sequence type, and contains the string type, the vector type, the bool-vector type, and the char-table type.

2.4.8 String Type

A *string* is an array of characters. Strings are used for many purposes in Emacs, as can be expected in a text editor; for example, as the names of Lisp symbols, as messages for the user, and to represent text extracted from buffers. Strings in Lisp are constants: evaluation of a string returns the same string.

See Chapter 4 [Strings and Characters], page 54, for functions that operate on strings.

2.4.8.1 Syntax for Strings

The read syntax for a string is a double-quote, an arbitrary number of characters, and another double-quote, `"like this"`. To include a double-quote in a string, precede it with a backslash; thus, `"\""` is a string containing just one double-quote character. Likewise, you can include a backslash by preceding it with another backslash, like this: `"this \\ is a single embedded backslash"`.

Since a string is an array of characters, you can specify the string characters using the read syntax of characters, but without the leading question mark. This is useful for including in string constants characters that don't stand for themselves. Thus, control characters can be specified as escape sequences that start with a backslash; for example, `"foo\r"` yields '`foo`' followed by the carriage return character. See Section 2.4.3.1 [Basic Char Syntax], page 11, for escape sequences of other control characters. Similarly, you can use the special read syntax for control characters (see Section 2.4.3.3 [Ctl-Char Syntax], page 13), as in `"foo\^Ibar"`, which produces a tab character embedded within a string. You can also use the escape sequences for non-ASCII characters described in Section 2.4.3.2 [General Escape Syntax], page 12, as in `"\N{LATIN SMALL LETTER A WITH GRAVE}"` and `"\u00e0"` (however, see a caveat with non-ASCII characters in Section 2.4.8.2 [Non-ASCII in Strings], page 20).

The newline character is not special in the read syntax for strings; if you write a new line between the double-quotes, it becomes a character in the string. But an escaped newline—one that is preceded by '\'—does not become part of the string; i.e., the Lisp reader ignores an escaped newline while reading a string. An escaped space '\ ' is likewise ignored.

```
"It is useful to include newlines
in documentation strings,
but the newline is \
ignored if escaped."
     ⇒ "It is useful to include newlines
in documentation strings,
but the newline is ignored if escaped."
```

2.4.8.2 Non-ASCII Characters in Strings

There are two text representations for non-ASCII characters in Emacs strings: multibyte and unibyte (see Section 34.1 [Text Representations], page 966). Roughly speaking, unibyte strings store raw bytes, while multibyte strings store human-readable text. Each character in a unibyte string is a byte, i.e., its value is between 0 and 255. By contrast, each character in a multibyte string may have a value between 0 to 4194303 (see Section 2.4.3 [Character Type], page 11). In both cases, characters above 127 are non-ASCII.

You can include a non-ASCII character in a string constant by writing it literally. If the string constant is read from a multibyte source, such as a multibyte buffer or string,

or a file that would be visited as multibyte, then Emacs reads each non-ASCII character as a multibyte character and automatically makes the string a multibyte string. If the string constant is read from a unibyte source, then Emacs reads the non-ASCII character as unibyte, and makes the string unibyte.

Instead of writing a character literally into a multibyte string, you can write it as its character code using an escape sequence. See Section 2.4.3.2 [General Escape Syntax], page 12, for details about escape sequences.

If you use any Unicode-style escape sequence '`\uNNNN`' or '`\UOONNNNNN`' in a string constant (even for an ASCII character), Emacs automatically assumes that it is multibyte.

You can also use hexadecimal escape sequences ('`\x`*n*') and octal escape sequences ('`\`*n*') in string constants. **But beware:** If a string constant contains octal escape sequences or one- or two-digit hexadecimal escape sequences, and these escape sequences all specify unibyte characters (i.e., codepoints less than 256), and there are no other literal non-ASCII characters or Unicode-style escape sequences in the string, then Emacs automatically assumes that it is a unibyte string. That is to say, it assumes that all non-ASCII characters occurring in the string are 8-bit raw bytes.

In hexadecimal and octal escape sequences, the escaped character code may contain a variable number of digits, so the first subsequent character which is not a valid hexadecimal or octal digit terminates the escape sequence. If the next character in a string could be interpreted as a hexadecimal or octal digit, write '`\ `' (backslash and space) to terminate the escape sequence. For example, '`\xe0\ `' represents one character, '`a`' with grave accent. '`\ `' in a string constant is just like backslash-newline; it does not contribute any character to the string, but it does terminate any preceding hex escape.

2.4.8.3 Nonprinting Characters in Strings

You can use the same backslash escape-sequences in a string constant as in character literals (but do not use the question mark that begins a character constant). For example, you can write a string containing the nonprinting characters tab and `C-a`, with commas and spaces between them, like this: `"\t, \C-a"`. See Section 2.4.3 [Character Type], page 11, and its subsections for a description of the various kinds of read syntax for characters.

However, not all of the characters you can write with backslash escape-sequences are valid in strings. The only control characters that a string can hold are the ASCII control characters. Strings do not distinguish case in ASCII control characters.

Properly speaking, strings cannot hold meta characters; but when a string is to be used as a key sequence, there is a special convention that provides a way to represent meta versions of ASCII characters in a string. If you use the '`\M-`' syntax to indicate a meta character in a string constant, this sets the 2^7 bit of the character in the string. If the string is used in `define-key` or `lookup-key`, this numeric code is translated into the equivalent meta character. See Section 2.4.3 [Character Type], page 11.

Strings cannot hold characters that have the hyper, super, or alt modifiers.

2.4.8.4 Text Properties in Strings

A string can hold properties for the characters it contains, in addition to the characters themselves. This enables programs that copy text between strings and buffers to copy the text's properties with no special effort. See Section 33.19 [Text Properties], page 919, for an

explanation of what text properties mean. Strings with text properties use a special read and print syntax:

```
#("characters" property-data...)
```

where *property-data* consists of zero or more elements, in groups of three as follows:

```
beg end plist
```

The elements *beg* and *end* are integers, and together specify a range of indices in the string; *plist* is the property list for that range. For example,

```
#("foo bar" 0 3 (face bold) 3 4 nil 4 7 (face italic))
```

represents a string whose textual contents are '`foo bar`', in which the first three characters have a `face` property with value `bold`, and the last three have a `face` property with value `italic`. (The fourth character has no text properties, so its property list is `nil`. It is not actually necessary to mention ranges with `nil` as the property list, since any characters not mentioned in any range will default to having no properties.)

2.4.9 Vector Type

A *vector* is a one-dimensional array of elements of any type. It takes a constant amount of time to access any element of a vector. (In a list, the access time of an element is proportional to the distance of the element from the beginning of the list.)

The printed representation of a vector consists of a left square bracket, the elements, and a right square bracket. This is also the read syntax. Like numbers and strings, vectors are considered constants for evaluation.

```
[1 "two" (three)]      ; A vector of three elements.
     ⇒ [1 "two" (three)]
```

See Section 6.4 [Vectors], page 117, for functions that work with vectors.

2.4.10 Char-Table Type

A *char-table* is a one-dimensional array of elements of any type, indexed by character codes. Char-tables have certain extra features to make them more useful for many jobs that involve assigning information to character codes—for example, a char-table can have a parent to inherit from, a default value, and a small number of extra slots to use for special purposes. A char-table can also specify a single value for a whole character set.

The printed representation of a char-table is like a vector except that there is an extra '`#^`' at the beginning.[1]

See Section 6.6 [Char-Tables], page 119, for special functions to operate on char-tables. Uses of char-tables include:

- Case tables (see Section 4.10 [Case Tables], page 74).
- Character category tables (see Section 36.8 [Categories], page 1039).
- Display tables (see Section 41.23.2 [Display Tables], page 1248).
- Syntax tables (see Chapter 36 [Syntax Tables], page 1025).

[1] You may also encounter '`#^^`', used for sub-char-tables.

2.4.11 Bool-Vector Type

A *bool-vector* is a one-dimensional array whose elements must be `t` or `nil`.

The printed representation of a bool-vector is like a string, except that it begins with '`#&`' followed by the length. The string constant that follows actually specifies the contents of the bool-vector as a bitmap—each character in the string contains 8 bits, which specify the next 8 elements of the bool-vector (1 stands for `t`, and 0 for `nil`). The least significant bits of the character correspond to the lowest indices in the bool-vector.

```
(make-bool-vector 3 t)
     ⇒ #&3"^G"
(make-bool-vector 3 nil)
     ⇒ #&3"^@"
```

These results make sense, because the binary code for '`C-g`' is 111 and '`C-@`' is the character with code 0.

If the length is not a multiple of 8, the printed representation shows extra elements, but these extras really make no difference. For instance, in the next example, the two bool-vectors are equal, because only the first 3 bits are used:

```
(equal #&3"\377" #&3"\007")
     ⇒ t
```

2.4.12 Hash Table Type

A hash table is a very fast kind of lookup table, somewhat like an alist in that it maps keys to corresponding values, but much faster. The printed representation of a hash table specifies its properties and contents, like this:

```
(make-hash-table)
     ⇒ #s(hash-table size 65 test eql rehash-size 1.5
                             rehash-threshold 0.8125 data ())
```

See Chapter 8 [Hash Tables], page 127, for more information about hash tables.

2.4.13 Function Type

Lisp functions are executable code, just like functions in other programming languages. In Lisp, unlike most languages, functions are also Lisp objects. A non-compiled function in Lisp is a lambda expression: that is, a list whose first element is the symbol `lambda` (see Section 13.2 [Lambda Expressions], page 232).

In most programming languages, it is impossible to have a function without a name. In Lisp, a function has no intrinsic name. A lambda expression can be called as a function even though it has no name; to emphasize this, we also call it an *anonymous function* (see Section 13.7 [Anonymous Functions], page 243). A named function in Lisp is just a symbol with a valid function in its function cell (see Section 13.4 [Defining Functions], page 237).

Most of the time, functions are called when their names are written in Lisp expressions in Lisp programs. However, you can construct or obtain a function object at run time and then call it with the primitive functions `funcall` and `apply`. See Section 13.5 [Calling Functions], page 239.

2.4.14 Macro Type

A *Lisp macro* is a user-defined construct that extends the Lisp language. It is represented as an object much like a function, but with different argument-passing semantics. A Lisp macro has the form of a list whose first element is the symbol `macro` and whose CDR is a Lisp function object, including the `lambda` symbol.

Lisp macro objects are usually defined with the built-in `defmacro` macro, but any list that begins with `macro` is a macro as far as Emacs is concerned. See Chapter 14 [Macros], page 269, for an explanation of how to write a macro.

Warning: Lisp macros and keyboard macros (see Section 22.16 [Keyboard Macros], page 479) are entirely different things. When we use the word "macro" without qualification, we mean a Lisp macro, not a keyboard macro.

2.4.15 Primitive Function Type

A *primitive function* is a function callable from Lisp but written in the C programming language. Primitive functions are also called *subrs* or *built-in functions*. (The word "subr" is derived from "subroutine".) Most primitive functions evaluate all their arguments when they are called. A primitive function that does not evaluate all its arguments is called a *special form* (see Section 10.1.7 [Special Forms], page 149).

It does not matter to the caller of a function whether the function is primitive. However, this does matter if you try to redefine a primitive with a function written in Lisp. The reason is that the primitive function may be called directly from C code. Calls to the redefined function from Lisp will use the new definition, but calls from C code may still use the built-in definition. Therefore, **we discourage redefinition of primitive functions**.

The term *function* refers to all Emacs functions, whether written in Lisp or C. See Section 2.4.13 [Function Type], page 23, for information about the functions written in Lisp.

Primitive functions have no read syntax and print in hash notation with the name of the subroutine.

```
(symbol-function 'car)          ; Access the function cell
                                ;    of the symbol.
     ⇒ #<subr car>
(subrp (symbol-function 'car))  ; Is this a primitive function?
     ⇒ t                        ; Yes.
```

2.4.16 Byte-Code Function Type

Byte-code function objects are produced by byte-compiling Lisp code (see Chapter 17 [Byte Compilation], page 317). Internally, a byte-code function object is much like a vector; however, the evaluator handles this data type specially when it appears in a function call. See Section 17.7 [Byte-Code Objects], page 324.

The printed representation and read syntax for a byte-code function object is like that for a vector, with an additional '`#`' before the opening '`[`'.

2.4.17 Record Type

A *record* is much like a `vector`. However, the first element is used to hold its type as returned by `type-of`. The purpose of records is to allow programmers to create objects with new types that are not built into Emacs.

See Chapter 7 [Records], page 125, for functions that work with records.

2.4.18 Type Descriptors

A *type descriptor* is a `record` which holds information about a type. Slot 1 in the record must be a symbol naming the type, and `type-of` relies on this to return the type of `record` objects. No other type descriptor slot is used by Emacs; they are free for use by Lisp extensions.

An example of a type descriptor is any instance of `cl-structure-class`.

2.4.19 Autoload Type

An *autoload object* is a list whose first element is the symbol `autoload`. It is stored as the function definition of a symbol, where it serves as a placeholder for the real definition. The autoload object says that the real definition is found in a file of Lisp code that should be loaded when necessary. It contains the name of the file, plus some other information about the real definition.

After the file has been loaded, the symbol should have a new function definition that is not an autoload object. The new definition is then called as if it had been there to begin with. From the user's point of view, the function call works as expected, using the function definition in the loaded file.

An autoload object is usually created with the function `autoload`, which stores the object in the function cell of a symbol. See Section 16.5 [Autoload], page 305, for more details.

2.4.20 Finalizer Type

A *finalizer object* helps Lisp code clean up after objects that are no longer needed. A finalizer holds a Lisp function object. When a finalizer object becomes unreachable after a garbage collection pass, Emacs calls the finalizer's associated function object. When deciding whether a finalizer is reachable, Emacs does not count references from finalizer objects themselves, allowing you to use finalizers without having to worry about accidentally capturing references to finalized objects themselves.

Errors in finalizers are printed to `*Messages*`. Emacs runs a given finalizer object's associated function exactly once, even if that function fails.

`make-finalizer` *function* [Function]

> Make a finalizer that will run *function*. *function* will be called after garbage collection when the returned finalizer object becomes unreachable. If the finalizer object is reachable only through references from finalizer objects, it does not count as reachable for the purpose of deciding whether to run *function*. *function* will be run once per finalizer object.

2.5 Editing Types

The types in the previous section are used for general programming purposes, and most of them are common to most Lisp dialects. Emacs Lisp provides several additional data types for purposes connected with editing.

2.5.1 Buffer Type

A *buffer* is an object that holds text that can be edited (see Chapter 28 [Buffers], page 673). Most buffers hold the contents of a disk file (see Chapter 26 [Files], page 610) so they can be edited, but some are used for other purposes. Most buffers are also meant to be seen by the user, and therefore displayed, at some time, in a window (see Chapter 29 [Windows], page 692). But a buffer need not be displayed in any window. Each buffer has a designated position called *point* (see Chapter 31 [Positions], page 855); most editing commands act on the contents of the current buffer in the neighborhood of point. At any time, one buffer is the *current buffer*.

The contents of a buffer are much like a string, but buffers are not used like strings in Emacs Lisp, and the available operations are different. For example, you can insert text efficiently into an existing buffer, altering the buffer's contents, whereas inserting text into a string requires concatenating substrings, and the result is an entirely new string object.

Many of the standard Emacs functions manipulate or test the characters in the current buffer; a whole chapter in this manual is devoted to describing these functions (see Chapter 33 [Text], page 880).

Several other data structures are associated with each buffer:

- a local syntax table (see Chapter 36 [Syntax Tables], page 1025);
- a local keymap (see Chapter 23 [Keymaps], page 481); and,
- a list of buffer-local variable bindings (see Section 12.11 [Buffer-Local Variables], page 207).
- overlays (see Section 41.9 [Overlays], page 1154).
- text properties for the text in the buffer (see Section 33.19 [Text Properties], page 919).

The local keymap and variable list contain entries that individually override global bindings or values. These are used to customize the behavior of programs in different buffers, without actually changing the programs.

A buffer may be *indirect*, which means it shares the text of another buffer, but presents it differently. See Section 28.11 [Indirect Buffers], page 689.

Buffers have no read syntax. They print in hash notation, showing the buffer name.

```
(current-buffer)
     ⇒ #<buffer objects.texi>
```

2.5.2 Marker Type

A *marker* denotes a position in a specific buffer. Markers therefore have two components: one for the buffer, and one for the position. Changes in the buffer's text automatically relocate the position value as necessary to ensure that the marker always points between the same two characters in the buffer.

Markers have no read syntax. They print in hash notation, giving the current character position and the name of the buffer.

```
(point-marker)
     ⇒ #<marker at 10779 in objects.texi>
```

See Chapter 32 [Markers], page 871, for information on how to test, create, copy, and move markers.

2.5.3 Window Type

A *window* describes the portion of the screen that Emacs uses to display buffers. Every live window (see Section 29.1 [Basic Windows], page 692) has one associated buffer, whose contents appear in that window. By contrast, a given buffer may appear in one window, no window, or several windows. Windows are grouped on the screen into frames; each window belongs to one and only one frame. See Section 2.5.4 [Frame Type], page 27.

Though many windows may exist simultaneously, at any time one window is designated the *selected window* (see Section 29.3 [Selecting Windows], page 698). This is the window where the cursor is (usually) displayed when Emacs is ready for a command. The selected window usually displays the current buffer (see Section 28.2 [Current Buffer], page 673), but this is not necessarily the case.

Windows have no read syntax. They print in hash notation, giving the window number and the name of the buffer being displayed. The window numbers exist to identify windows uniquely, since the buffer displayed in any given window can change frequently.

```
(selected-window)
     ⇒ #<window 1 on objects.texi>
```

See Chapter 29 [Windows], page 692, for a description of the functions that work on windows.

2.5.4 Frame Type

A *frame* is a screen area that contains one or more Emacs windows; we also use the term "frame" to refer to the Lisp object that Emacs uses to refer to the screen area.

Frames have no read syntax. They print in hash notation, giving the frame's title, plus its address in core (useful to identify the frame uniquely).

```
(selected-frame)
     ⇒ #<frame emacs@psilocin.gnu.org 0xdac80>
```

See Chapter 30 [Frames], page 787, for a description of the functions that work on frames.

2.5.5 Terminal Type

A *terminal* is a device capable of displaying one or more Emacs frames (see Section 2.5.4 [Frame Type], page 27).

Terminals have no read syntax. They print in hash notation giving the terminal's ordinal number and its TTY device file name.

```
(get-device-terminal nil)
     ⇒ #<terminal 1 on /dev/tty>
```

2.5.6 Window Configuration Type

A *window configuration* stores information about the positions, sizes, and contents of the windows in a frame, so you can recreate the same arrangement of windows later.

Window configurations do not have a read syntax; their print syntax looks like '`#<window-configuration>`'. See Section 29.26 [Window Configurations], page 777, for a description of several functions related to window configurations.

2.5.7 Frame Configuration Type

A *frame configuration* stores information about the positions, sizes, and contents of the windows in all frames. It is not a primitive type—it is actually a list whose CAR is `frame-configuration` and whose CDR is an alist. Each alist element describes one frame, which appears as the CAR of that element.

See Section 30.13 [Frame Configurations], page 833, for a description of several functions related to frame configurations.

2.5.8 Process Type

The word *process* usually means a running program. Emacs itself runs in a process of this sort. However, in Emacs Lisp, a process is a Lisp object that designates a subprocess created by the Emacs process. Programs such as shells, GDB, ftp, and compilers, running in subprocesses of Emacs, extend the capabilities of Emacs. An Emacs subprocess takes textual input from Emacs and returns textual output to Emacs for further manipulation. Emacs can also send signals to the subprocess.

Process objects have no read syntax. They print in hash notation, giving the name of the process:

```
(process-list)
     ⇒ (#<process shell>)
```

See Chapter 40 [Processes], page 1081, for information about functions that create, delete, return information about, send input or signals to, and receive output from processes.

2.5.9 Thread Type

A *thread* in Emacs represents a separate thread of Emacs Lisp execution. It runs its own Lisp program, has its own current buffer, and can have subprocesses locked to it, i.e. subprocesses whose output only this thread can accept. See Chapter 39 [Threads], page 1076.

Thread objects have no read syntax. They print in hash notation, giving the name of the thread (if it has been given a name) or its address in core:

```
(all-threads)
    ⇒ (#<thread 0176fc40>)
```

2.5.10 Mutex Type

A *mutex* is an exclusive lock that threads can own and disown, in order to synchronize between them. See Section 39.2 [Mutexes], page 1078.

Mutex objects have no read syntax. They print in hash notation, giving the name of the mutex (if it has been given a name) or its address in core:

```
(make-mutex "my-mutex")
    ⇒ #<mutex my-mutex>
(make-mutex)
    ⇒ #<mutex 01c7e4e0>
```

2.5.11 Condition Variable Type

A *condition variable* is a device for a more complex thread synchronization than the one supported by a mutex. A thread can wait on a condition variable, to be woken up when some other thread notifies the condition.

Condition variable objects have no read syntax. They print in hash notation, giving the name of the condition variable (if it has been given a name) or its address in core:

```
(make-condition-variable (make-mutex))
    ⇒ #<condvar 01c45ae8>
```

2.5.12 Stream Type

A *stream* is an object that can be used as a source or sink for characters—either to supply characters for input or to accept them as output. Many different types can be used this way: markers, buffers, strings, and functions. Most often, input streams (character sources) obtain characters from the keyboard, a buffer, or a file, and output streams (character sinks) send characters to a buffer, such as a `*Help*` buffer, or to the echo area.

The object `nil`, in addition to its other meanings, may be used as a stream. It stands for the value of the variable `standard-input` or `standard-output`. Also, the object `t` as a stream specifies input using the minibuffer (see Chapter 21 [Minibuffers], page 386) or output in the echo area (see Section 41.4 [The Echo Area], page 1135).

Streams have no special printed representation or read syntax, and print as whatever primitive type they are.

See Chapter 20 [Read and Print], page 372, for a description of functions related to streams, including parsing and printing functions.

2.5.13 Keymap Type

A *keymap* maps keys typed by the user to commands. This mapping controls how the user's command input is executed. A keymap is actually a list whose CAR is the symbol `keymap`.

See Chapter 23 [Keymaps], page 481, for information about creating keymaps, handling prefix keys, local as well as global keymaps, and changing key bindings.

2.5.14 Overlay Type

An *overlay* specifies properties that apply to a part of a buffer. Each overlay applies to a specified range of the buffer, and contains a property list (a list whose elements are alternating property names and values). Overlay properties are used to present parts of the buffer temporarily in a different display style. Overlays have no read syntax, and print in hash notation, giving the buffer name and range of positions.

See Section 41.9 [Overlays], page 1154, for information on how you can create and use overlays.

2.5.15 Font Type

A *font* specifies how to display text on a graphical terminal. There are actually three separate font types—*font objects*, *font specs*, and *font entities*—each of which has slightly different properties. None of them have a read syntax; their print syntax looks like '`#<font-object>`', '`#<font-spec>`', and '`#<font-entity>`' respectively. See Section 41.12.12 [Low-Level Font], page 1188, for a description of these Lisp objects.

2.5.16 Xwidget Type

An *xwidget* is a special display element, such as a web browser, that can be embedded inside a buffer. Each window that displays an xwidget will also have an *xwidget view*, which on X-Windows corresponds to a single X window used to display the widget.

Neither of these objects are readable; their print syntaxes look like '`#<xwidget>`' and '`#<xwidget-view>`', respectively. See Section 41.19 [Xwidgets], page 1233, for a more detailed description of xwidgets.

2.6 Read Syntax for Circular Objects

To represent shared or circular structures within a complex of Lisp objects, you can use the reader constructs '`#n=`' and '`#n#`'.

Use `#n=` before an object to label it for later reference; subsequently, you can use `#n#` to refer the same object in another place. Here, *n* is some integer. For example, here is how to make a list in which the first element recurs as the third element:

```
(#1=(a) b #1#)
```

This differs from ordinary syntax such as this

```
((a) b (a))
```

which would result in a list whose first and third elements look alike but are not the same Lisp object. This shows the difference:

```
(prog1 nil
  (setq x '(#1=(a) b #1#)))
(eq (nth 0 x) (nth 2 x))
     ⇒ t
(setq x '((a) b (a)))
(eq (nth 0 x) (nth 2 x))
     ⇒ nil
```

You can also use the same syntax to make a circular structure, which appears as an element within itself. Here is an example:

```
#1=(a #1#)
```

This makes a list whose second element is the list itself. Here's how you can see that it really works:

```
(prog1 nil
  (setq x '#1=(a #1#)))
(eq x (cadr x))
     ⇒ t
```

The Lisp printer can produce this syntax to record circular and shared structure in a Lisp object, if you bind the variable `print-circle` to a non-`nil` value. See Section 20.6 [Output Variables], page 381.

2.7 Type Predicates

The Emacs Lisp interpreter itself does not perform type checking on the actual arguments passed to functions when they are called. It could not do so, since function arguments in Lisp do not have declared data types, as they do in other programming languages. It is therefore up to the individual function to test whether each actual argument belongs to a type that the function can use.

All built-in functions do check the types of their actual arguments when appropriate, and signal a `wrong-type-argument` error if an argument is of the wrong type. For example, here is what happens if you pass an argument to `+` that it cannot handle:

```
(+ 2 'a)
     error  Wrong type argument: number-or-marker-p, a
```

If you want your program to handle different types differently, you must do explicit type checking. The most common way to check the type of an object is to call a *type predicate* function. Emacs has a type predicate for each type, as well as some predicates for combinations of types.

A type predicate function takes one argument; it returns `t` if the argument belongs to the appropriate type, and `nil` otherwise. Following a general Lisp convention for predicate functions, most type predicates' names end with '`p`'.

Here is an example which uses the predicates `listp` to check for a list and `symbolp` to check for a symbol.

```
(defun add-on (x)
  (cond ((symbolp x)
         ;; If X is a symbol, put it on LIST.
         (setq list (cons x list)))
        ((listp x)
         ;; If X is a list, add its elements to LIST.
         (setq list (append x list)))
        (t
         ;; We handle only symbols and lists.
         (error "Invalid argument %s in add-on" x))))
```

Here is a table of predefined type predicates, in alphabetical order, with references to further information.

`atom` See Section 5.2 [List-related Predicates], page 78.

`arrayp` See Section 6.3 [Array Functions], page 116.

`bignump` See Section 3.3 [Predicates on Numbers], page 41.

`bool-vector-p`
See Section 6.7 [Bool-Vectors], page 121.

`booleanp` See Section 1.3.2 [nil and t], page 2.

`bufferp` See Section 28.1 [Buffer Basics], page 673.

`byte-code-function-p`
See Section 2.4.16 [Byte-Code Type], page 24.

`compiled-function-p`
See Section 2.4.16 [Byte-Code Type], page 24.

`case-table-p`
See Section 4.10 [Case Tables], page 74.

`char-or-string-p`
See Section 4.2 [Predicates for Strings], page 55.

`char-table-p`
See Section 6.6 [Char-Tables], page 119.

`commandp` See Section 22.3 [Interactive Call], page 433.

`condition-variable-p`
See Section 39.3 [Condition Variables], page 1078.

`consp` See Section 5.2 [List-related Predicates], page 78.

`custom-variable-p`
See Section 15.3 [Variable Definitions], page 281.

`fixnump` See Section 3.3 [Predicates on Numbers], page 41.

`floatp` See Section 3.3 [Predicates on Numbers], page 41.

`fontp` See Section 41.12.12 [Low-Level Font], page 1188.

`frame-configuration-p`
See Section 30.13 [Frame Configurations], page 833.

`frame-live-p`
See Section 30.7 [Deleting Frames], page 824.

`framep` See Chapter 30 [Frames], page 787.

`functionp`
See Chapter 13 [Functions], page 230.

`hash-table-p`
See Section 8.4 [Other Hash], page 131.

`integer-or-marker-p`
See Section 32.2 [Predicates on Markers], page 872.

`integerp` See Section 3.3 [Predicates on Numbers], page 41.

`keymapp` See Section 23.4 [Creating Keymaps], page 484.

`keywordp` See Section 12.2 [Constant Variables], page 189.

`listp` See Section 5.2 [List-related Predicates], page 78.

`markerp` See Section 32.2 [Predicates on Markers], page 872.

`mutexp` See Section 39.2 [Mutexes], page 1078.

`nlistp` See Section 5.2 [List-related Predicates], page 78.

`number-or-marker-p`
See Section 32.2 [Predicates on Markers], page 872.

`numberp` See Section 3.3 [Predicates on Numbers], page 41.

`overlayp` See Section 41.9 [Overlays], page 1154.

`processp` See Chapter 40 [Processes], page 1081.

`recordp` See Section 2.4.17 [Record Type], page 25.

`sequencep`
See Section 6.1 [Sequence Functions], page 102.

`string-or-null-p`
See Section 4.2 [Predicates for Strings], page 55.

`stringp` See Section 4.2 [Predicates for Strings], page 55.

`subrp` See Section 13.9 [Function Cells], page 248.

`symbolp` See Chapter 9 [Symbols], page 133.

`syntax-table-p`
See Chapter 36 [Syntax Tables], page 1025.

`threadp` See Section 39.1 [Basic Thread Functions], page 1076.

`vectorp` See Section 6.4 [Vectors], page 117.

`wholenump`
See Section 3.3 [Predicates on Numbers], page 41.

`window-configuration-p`
See Section 29.26 [Window Configurations], page 777.

`window-live-p`
See Section 29.8 [Deleting Windows], page 713.

`windowp` See Section 29.1 [Basic Windows], page 692.

The most general way to check the type of an object is to call the function `type-of`. Recall that each object belongs to one and only one primitive type; `type-of` tells you which one (see Chapter 2 [Lisp Data Types], page 8). But `type-of` knows nothing about non-primitive types. In most cases, it is more convenient to use type predicates than `type-of`.

`type-of` *object* [Function]

This function returns a symbol naming the primitive type of *object*. The value is one of the symbols `bool-vector`, `buffer`, `char-table`, `compiled-function`, `condition-variable`, `cons`, `finalizer`, `float`, `font-entity`, `font-object`, `font-spec`, `frame`, `hash-table`, `integer`, `marker`, `mutex`, `overlay`, `process`, `string`, `subr`, `symbol`, `thread`, `vector`, `window`, or `window-configuration`. However, if *object* is a record, the type specified by its first slot is returned; Chapter 7 [Records], page 125.

```
(type-of 1)
```

```
     ⇒ integer
(type-of 'nil)
     ⇒ symbol
(type-of '())    ; () is nil.
     ⇒ symbol
(type-of '(x))
     ⇒ cons
(type-of (record 'foo))
     ⇒ foo
```

2.8 Equality Predicates

Here we describe functions that test for equality between two objects. Other functions test equality of contents between objects of specific types, e.g., strings. For these predicates, see the appropriate chapter describing the data type.

`eq` *object1 object2* [Function]

This function returns `t` if *object1* and *object2* are the same object, and `nil` otherwise.

If *object1* and *object2* are symbols with the same name, they are normally the same object—but see Section 9.3 [Creating Symbols], page 135, for exceptions. For other non-numeric types (e.g., lists, vectors, strings), two arguments with the same contents or elements are not necessarily `eq` to each other: they are `eq` only if they are the same object, meaning that a change in the contents of one will be reflected by the same change in the contents of the other.

If *object1* and *object2* are numbers with differing types or values, then they cannot be the same object and `eq` returns `nil`. If they are fixnums with the same value, then they are the same object and `eq` returns `t`. If they were computed separately but happen to have the same value and the same non-fixnum numeric type, then they might or might not be the same object, and `eq` returns `t` or `nil` depending on whether the Lisp interpreter created one object or two.

```
(eq 'foo 'foo)
     ⇒ t

(eq ?A ?A)
     ⇒ t

(eq 3.0 3.0)
     ⇒ t or nil
;; Equal floats may or may not be the same object.

(eq (make-string 3 ?A) (make-string 3 ?A))
     ⇒ nil

(eq "asdf" "asdf")
     ⇒ t or nil
;; Equal string constants or may not be the same object.
```

```
(eq '(1 (2 (3))) '(1 (2 (3))))
     ⇒ nil

(setq foo '(1 (2 (3))))
     ⇒ (1 (2 (3)))
(eq foo foo)
     ⇒ t
(eq foo '(1 (2 (3))))
     ⇒ nil

(eq [(1 2) 3] [(1 2) 3])
     ⇒ nil

(eq (point-marker) (point-marker))
     ⇒ nil
```

The `make-symbol` function returns an uninterned symbol, distinct from the symbol that is used if you write the name in a Lisp expression. Distinct symbols with the same name are not `eq`. See Section 9.3 [Creating Symbols], page 135.

```
(eq (make-symbol "foo") 'foo)
     ⇒ nil
```

The Emacs Lisp byte compiler may collapse identical literal objects, such as literal strings, into references to the same object, with the effect that the byte-compiled code will compare such objects as `eq`, while the interpreted version of the same code will not. Therefore, your code should never rely on objects with the same literal contents being either `eq` or not `eq`, it should instead use functions that compare object contents such as `equal`, described below. Similarly, your code should not modify literal objects (e.g., put text properties on literal strings), since doing that might affect other literal objects of the same contents, if the byte compiler collapses them.

`equal` *object1 object2* [Function]

This function returns `t` if *object1* and *object2* have equal components, and `nil` otherwise. Whereas `eq` tests if its arguments are the same object, `equal` looks inside nonidentical arguments to see if their elements or contents are the same. So, if two objects are `eq`, they are `equal`, but the converse is not always true.

```
(equal 'foo 'foo)
     ⇒ t

(equal 456 456)
     ⇒ t

(equal "asdf" "asdf")
     ⇒ t
(eq "asdf" "asdf")
     ⇒ nil

(equal '(1 (2 (3))) '(1 (2 (3))))
     ⇒ t
```

```
(eq '(1 (2 (3))) '(1 (2 (3))))
     ⇒ nil

(equal [(1 2) 3] [(1 2) 3])
     ⇒ t
(eq [(1 2) 3] [(1 2) 3])
     ⇒ nil

(equal (point-marker) (point-marker))
     ⇒ t

(eq (point-marker) (point-marker))
     ⇒ nil
```

Comparison of strings is case-sensitive, but does not take account of text properties—it compares only the characters in the strings. See Section 33.19 [Text Properties], page 919. Use `equal-including-properties` to also compare text properties. For technical reasons, a unibyte string and a multibyte string are `equal` if and only if they contain the same sequence of character codes and all these codes are in the range 0 through 127 (ASCII).

```
(equal "asdf" "ASDF")
     ⇒ nil
```

The `equal` function recursively compares the contents of objects if they are integers, strings, markers, vectors, bool-vectors, byte-code function objects, char-tables, records, or font objects. Other objects are considered `equal` only if they are `eq`. For example, two distinct buffers are never considered `equal`, even if their textual contents are the same.

For `equal`, equality is defined recursively; for example, given two cons cells *x* and *y*, `(equal` *x y*`)` returns `t` if and only if both the expressions below return `t`:

```
(equal (car x) (car y))
(equal (cdr x) (cdr y))
```

Comparing circular lists may therefore cause deep recursion that leads to an error, and this may result in counterintuitive behavior such as `(equal a b)` returning `t` whereas `(equal b a)` signals an error.

`equal-including-properties` *object1 object2* [Function]

This function behaves like `equal` in all cases but also requires that for two strings to be equal, they have the same text properties.

```
(equal "asdf" (propertize "asdf" 'asdf t))
     ⇒ t
(equal-including-properties "asdf"
                            (propertize "asdf" 'asdf t))
     ⇒ nil
```

2.9 Mutability

Some Lisp objects should never change. For example, the Lisp expression `"aaa"` yields a string, but you should not change its contents. And some objects cannot be changed; for example, although you can create a new number by calculating one, Lisp provides no operation to change the value of an existing number.

Other Lisp objects are *mutable*: it is safe to change their values via destructive operations involving side effects. For example, an existing marker can be changed by moving the marker to point to somewhere else.

Although numbers never change and all markers are mutable, some types have members some of which are mutable and others not. These types include conses, vectors, and strings. For example, although `"cons"` and `(symbol-name 'cons)` both yield strings that should not be changed, `(copy-sequence "cons")` and `(make-string 3 ?a)` both yield mutable strings that can be changed via later calls to `aset`.

A mutable object stops being mutable if it is part of an expression that is evaluated. For example:

```
(let* ((x (list 0.5))
       (y (eval (list 'quote x))))
  (setcar x 1.5) ;; The program should not do this.
  y)
```

Although the list `(0.5)` was mutable when it was created, it should not have been changed via `setcar` because it was given to `eval`. The reverse does not occur: an object that should not be changed never becomes mutable afterwards.

If a program attempts to change objects that should not be changed, the resulting behavior is undefined: the Lisp interpreter might signal an error, or it might crash or behave unpredictably in other ways.[2]

When similar constants occur as parts of a program, the Lisp interpreter might save time or space by reusing existing constants or their components. For example, `(eq "abc" "abc")` returns `t` if the interpreter creates only one instance of the string literal `"abc"`, and returns `nil` if it creates two instances. Lisp programs should be written so that they work regardless of whether this optimization is in use.

[2] This is the behavior specified for languages like Common Lisp and C for constants, and this differs from languages like JavaScript and Python where an interpreter is required to signal an error if a program attempts to change an immutable object. Ideally the Emacs Lisp interpreter will evolve in latter direction.

3 Numbers

GNU Emacs supports two numeric data types: *integers* and *floating-point numbers*. Integers are whole numbers such as −3, 0, 7, 13, and 511. Floating-point numbers are numbers with fractional parts, such as −4.5, 0.0, and 2.71828. They can also be expressed in exponential notation: '`1.5e2`' is the same as '`150.0`'; here, '`e2`' stands for ten to the second power, and that is multiplied by 1.5. Integer computations are exact. Floating-point computations often involve rounding errors, as the numbers have a fixed amount of precision.

3.1 Integer Basics

The Lisp reader reads an integer as a nonempty sequence of decimal digits with optional initial sign and optional final period.

```
 1          ; The integer 1.
 1.         ; The integer 1.
+1          ; Also the integer 1.
-1          ; The integer −1.
 0          ; The integer 0.
-0          ; The integer 0.
```

The syntax for integers in bases other than 10 consists of '`#`' followed by a radix indication followed by one or more digits. The radix indications are '`b`' for binary, '`o`' for octal, '`x`' for hex, and '*`radix`*`r`' for radix *radix*. Thus, '`#b`*`integer`*' reads *integer* in binary, and '`#`*`radix`*`r`*`integer`*' reads *integer* in radix *radix*. Allowed values of *radix* run from 2 to 36, and allowed digits are the first *radix* characters taken from '`0`'–'`9`', '`A`'–'`Z`'. Letter case is ignored and there is no initial sign or final period. For example:

```
#b101100 ⇒ 44
#o54 ⇒ 44
#x2c ⇒ 44
#24r1k ⇒ 44
```

To understand how various functions work on integers, especially the bitwise operators (see Section 3.8 [Bitwise Operations], page 48), it is often helpful to view the numbers in their binary form.

In binary, the decimal integer 5 looks like this:

```
...000101
```

(The ellipsis '`...`' stands for a conceptually infinite number of bits that match the leading bit; here, an infinite number of 0 bits. Later examples also use this '`...`' notation.)

The integer −1 looks like this:

```
...111111
```

−1 is represented as all ones. (This is called *two's complement* notation.)

Subtracting 4 from −1 returns the negative integer −5. In binary, the decimal integer 4 is 100. Consequently, −5 looks like this:

```
...111011
```

Many of the functions described in this chapter accept markers for arguments in place of numbers. (See Chapter 32 [Markers], page 871.) Since the actual arguments to such

functions may be either numbers or markers, we often give these arguments the name *number-or-marker*. When the argument value is a marker, its position value is used and its buffer is ignored.

In Emacs Lisp, text characters are represented by integers. Any integer between zero and the value of `(max-char)`, inclusive, is considered to be valid as a character. See Section 34.5 [Character Codes], page 970.

Integers in Emacs Lisp are not limited to the machine word size. Under the hood, though, there are two kinds of integers: smaller ones, called *fixnums*, and larger ones, called *bignums*. Although Emacs Lisp code ordinarily should not depend on whether an integer is a fixnum or a bignum, older Emacs versions support only fixnums, some functions in Emacs still accept only fixnums, and older Emacs Lisp code may have trouble when given bignums. For example, while older Emacs Lisp code could safely compare integers for numeric equality with `eq`, the presence of bignums means that equality predicates like `eql` and `=` should now be used to compare integers.

The range of values for bignums is limited by the amount of main memory, by machine characteristics such as the size of the word used to represent a bignum's exponent, and by the `integer-width` variable. These limits are typically much more generous than the limits for fixnums. A bignum is never numerically equal to a fixnum; Emacs always represents an integer in fixnum range as a fixnum, not a bignum.

The range of values for a fixnum depends on the machine. The minimum range is −536,870,912 to 536,870,911 (30 bits; i.e., -2^{29} to $2^{29} - 1$), but many machines provide a wider range.

`most-positive-fixnum` [Variable]
: The value of this variable is the greatest "small" integer that Emacs Lisp can handle. Typical values are $2^{29} - 1$ on 32-bit and $2^{61} - 1$ on 64-bit platforms.

`most-negative-fixnum` [Variable]
: The value of this variable is the numerically least "small" integer that Emacs Lisp can handle. It is negative. Typical values are -2^{29} on 32-bit and -2^{61} on 64-bit platforms.

`integer-width` [Variable]
: The value of this variable is a nonnegative integer that controls whether Emacs signals a range error when a large integer would be calculated. Integers with absolute values less than 2^n, where n is this variable's value, do not signal a range error. Attempts to create larger integers typically signal a range error, although there might be no signal if a larger integer can be created cheaply. Setting this variable to a large number can be costly if a computation creates huge integers.

3.2 Floating-Point Basics

Floating-point numbers are useful for representing numbers that are not integral. The range of floating-point numbers is the same as the range of the C data type `double` on the machine you are using. On almost all computers supported by Emacs, this is IEEE binary64 floating point format, which is standardized by IEEE Std 754-2019 and is discussed further in David Goldberg's paper "What Every Computer Scientist Should Know

About Floating-Point Arithmetic (`https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html`)". On modern platforms, floating-point operations follow the IEEE-754 standard closely; however, results are not always rounded correctly on some systems, notably 32-bit x86.

On some old computer systems, Emacs may not use IEEE floating-point. We know of one such system on which Emacs runs correctly, but does not follow IEEE-754: the VAX running NetBSD using GCC 10.4.0, where the VAX '`D_Floating`' format is used instead. IBM System/370-derived mainframes and their XL/C compiler are also capable of utilizing a hexadecimal floating point format, but Emacs has not yet been built in such a configuration.

The read syntax for floating-point numbers requires either a decimal point, an exponent, or both. Optional signs ('`+`' or '`-`') precede the number and its exponent. For example, '`1500.0`', '`+15e2`', '`15.0e+2`', '`+1500000e-3`', and '`.15e4`' are five ways of writing a floating-point number whose value is 1500. They are all equivalent. Like Common Lisp, Emacs Lisp requires at least one digit after a decimal point in a floating-point number that does not have an exponent; '`1500.`' is an integer, not a floating-point number.

Emacs Lisp treats `-0.0` as numerically equal to ordinary zero with respect to numeric comparisons like `=`. This follows the IEEE floating-point standard, which says `-0.0` and `0.0` are numerically equal even though other operations can distinguish them.

The IEEE floating-point standard supports positive infinity and negative infinity as floating-point values. It also provides for a class of values called NaN, or "not a number"; numerical functions return such values in cases where there is no correct answer. For example, `(/ 0.0 0.0)` returns a NaN. A NaN is never numerically equal to any value, not even to itself. NaNs carry a sign and a significand, and non-numeric functions treat two NaNs as equal when their signs and significands agree. Significands of NaNs are machine-dependent, as are the digits in their string representation.

NaNs are not available on systems which do not use IEEE floating-point arithmetic; if the read syntax for a NaN is used on a VAX, for example, the reader signals an error.

When NaNs and signed zeros are involved, non-numeric functions like `eql`, `equal`, `sxhash-eql`, `sxhash-equal` and `gethash` determine whether values are indistinguishable, not whether they are numerically equal. For example, when x and y are the same NaN, `(equal x y)` returns `t` whereas `(= x y)` uses numeric comparison and returns `nil`; conversely, `(equal 0.0 -0.0)` returns `nil` whereas `(= 0.0 -0.0)` returns `t`.

Here are read syntaxes for these special floating-point values:

infinity '`1.0e+INF`' and '`-1.0e+INF`'

not-a-number
'`0.0e+NaN`' and '`-0.0e+NaN`'

The following functions are specialized for handling floating-point numbers:

`isnan` *x* [Function]
This predicate returns `t` if its floating-point argument is a NaN, `nil` otherwise.

`frexp` *x* [Function]
This function returns a cons cell `(`*s*` . `*e*`)`, where *s* and *e* are respectively the significand and exponent of the floating-point number *x*.

If x is finite, then s is a floating-point number between 0.5 (inclusive) and 1.0 (exclusive), e is an integer, and $x = s2^e$. If x is zero or infinity, then s is the same as x. If x is a NaN, then s is also a NaN. If x is zero, then e is 0.

`ldexp` *s e* [Function]
Given a numeric significand s and an integer exponent e, this function returns the floating point number $s2^e$.

`copysign` *x1 x2* [Function]
This function copies the sign of *x2* to the value of *x1*, and returns the result. *x1* and *x2* must be floating point.

`logb` *x* [Function]
This function returns the binary exponent of x. More precisely, if x is finite and nonzero, the value is the logarithm base 2 of $|x|$, rounded down to an integer. If x is zero or infinite, the value is infinity; if x is a NaN, the value is a NaN.

```
(logb 10)
     ⇒ 3
(logb 10.0e20)
     ⇒ 69
(logb 0)
     ⇒ -1.0e+INF
```

3.3 Type Predicates for Numbers

The functions in this section test for numbers, or for a specific type of number. The functions `integerp` and `floatp` can take any type of Lisp object as argument (they would not be of much use otherwise), but the `zerop` predicate requires a number as its argument. See also `integer-or-marker-p` and `number-or-marker-p`, in Section 32.2 [Predicates on Markers], page 872.

`bignump` *object* [Function]
This predicate tests whether its argument is a large integer, and returns `t` if so, `nil` otherwise. Unlike small integers, large integers can be `=` or `eql` even if they are not `eq`.

`fixnump` *object* [Function]
This predicate tests whether its argument is a small integer, and returns `t` if so, `nil` otherwise. Small integers can be compared with `eq`.

`floatp` *object* [Function]
This predicate tests whether its argument is floating point and returns `t` if so, `nil` otherwise.

`integerp` *object* [Function]
This predicate tests whether its argument is an integer, and returns `t` if so, `nil` otherwise.

`numberp` *object* [Function]
This predicate tests whether its argument is a number (either integer or floating point), and returns `t` if so, `nil` otherwise.

`natnump` *object* [Function]
: This predicate (whose name comes from the phrase "natural number") tests to see whether its argument is a nonnegative integer, and returns `t` if so, `nil` otherwise. 0 is considered non-negative.

  `wholenump` is a synonym for `natnump`.

`zerop` *number* [Function]
: This predicate tests whether its argument is zero, and returns `t` if so, `nil` otherwise. The argument must be a number.

  `(zerop x)` is equivalent to `(= x 0)`.

3.4 Comparison of Numbers

To test numbers for numerical equality, you should normally use `=` instead of non-numeric comparison predicates like `eq`, `eql` and `equal`. Distinct floating-point and large integer objects can be numerically equal. If you use `eq` to compare them, you test whether they are the same *object*; if you use `eql` or `equal`, you test whether their values are *indistinguishable*. In contrast, `=` uses numeric comparison, and sometimes returns `t` when a non-numeric comparison would return `nil` and vice versa. See Section 3.2 [Float Basics], page 39.

In Emacs Lisp, if two fixnums are numerically equal, they are the same Lisp object. That is, `eq` is equivalent to `=` on fixnums. It is sometimes convenient to use `eq` for comparing an unknown value with a fixnum, because `eq` does not report an error if the unknown value is not a number—it accepts arguments of any type. By contrast, `=` signals an error if the arguments are not numbers or markers. However, it is better programming practice to use `=` if you can, even for comparing integers.

Sometimes it is useful to compare numbers with `eql` or `equal`, which treat two numbers as equal if they have the same data type (both integers, or both floating point) and the same value. By contrast, `=` can treat an integer and a floating-point number as equal. See Section 2.8 [Equality Predicates], page 34.

There is another wrinkle: because floating-point arithmetic is not exact, it is often a bad idea to check for equality of floating-point values. Usually it is better to test for approximate equality. Here's a function to do this:

```
(defvar fuzz-factor 1.0e-6)
(defun approx-equal (x y)
  (or (= x y)
      (< (/ (abs (- x y))
            (max (abs x) (abs y)))
         fuzz-factor)))
```

`=` *number-or-marker* **&rest** *number-or-markers* [Function]
: This function tests whether all its arguments are numerically equal, and returns `t` if so, `nil` otherwise.

`eql` *value1 value2* [Function]
: This function acts like `eq` except when both arguments are numbers. It compares numbers by type and numeric value, so that `(eql 1.0 1)` returns `nil`, but `(eql 1.0 1.0)` and `(eql 1 1)` both return `t`. This can be used to compare large integers as well

as small ones. Floating-point values with the same sign, exponent and fraction are `eql`. This differs from numeric comparison: `(eql 0.0 -0.0)` returns `nil` and `(eql 0.0e+NaN 0.0e+NaN)` returns `t`, whereas `=` does the opposite.

`/=` *number-or-marker1 number-or-marker2* [Function]
This function tests whether its arguments are numerically equal, and returns `t` if they are not, and `nil` if they are.

`<` *number-or-marker* **&rest** *number-or-markers* [Function]
This function tests whether each argument is strictly less than the following argument. It returns `t` if so, `nil` otherwise.

`<=` *number-or-marker* **&rest** *number-or-markers* [Function]
This function tests whether each argument is less than or equal to the following argument. It returns `t` if so, `nil` otherwise.

`>` *number-or-marker* **&rest** *number-or-markers* [Function]
This function tests whether each argument is strictly greater than the following argument. It returns `t` if so, `nil` otherwise.

`>=` *number-or-marker* **&rest** *number-or-markers* [Function]
This function tests whether each argument is greater than or equal to the following argument. It returns `t` if so, `nil` otherwise.

`max` *number-or-marker* **&rest** *numbers-or-markers* [Function]
This function returns the largest of its arguments.

```
(max 20)
     ⇒ 20
(max 1 2.5)
     ⇒ 2.5
(max 1 3 2.5)
     ⇒ 3
```

`min` *number-or-marker* **&rest** *numbers-or-markers* [Function]
This function returns the smallest of its arguments.

```
(min -4 1)
     ⇒ -4
```

`abs` *number* [Function]
This function returns the absolute value of *number*.

3.5 Numeric Conversions

To convert an integer to floating point, use the function `float`.

`float` *number* [Function]
This returns *number* converted to floating point. If *number* is already floating point, `float` returns it unchanged.

There are four functions to convert floating-point numbers to integers; they differ in how they round. All accept an argument *number* and an optional argument *divisor*. Both arguments may be integers or floating-point numbers. *divisor* may also be `nil`. If *divisor* is `nil` or omitted, these functions convert *number* to an integer, or return it unchanged if it already is an integer. If *divisor* is non-`nil`, they divide *number* by *divisor* and convert the result to an integer. If *divisor* is zero (whether integer or floating point), Emacs signals an `arith-error` error.

`truncate` *number* **&optional** *divisor* [Function]

This returns *number*, converted to an integer by rounding towards zero.

```
(truncate 1.2)
     ⇒ 1
(truncate 1.7)
     ⇒ 1
(truncate -1.2)
     ⇒ -1
(truncate -1.7)
     ⇒ -1
```

`floor` *number* **&optional** *divisor* [Function]

This returns *number*, converted to an integer by rounding downward (towards negative infinity).

If *divisor* is specified, this uses the kind of division operation that corresponds to `mod`, rounding downward.

```
(floor 1.2)
     ⇒ 1
(floor 1.7)
     ⇒ 1
(floor -1.2)
     ⇒ -2
(floor -1.7)
     ⇒ -2
(floor 5.99 3)
     ⇒ 1
```

`ceiling` *number* **&optional** *divisor* [Function]

This returns *number*, converted to an integer by rounding upward (towards positive infinity).

```
(ceiling 1.2)
     ⇒ 2
(ceiling 1.7)
     ⇒ 2
(ceiling -1.2)
     ⇒ -1
(ceiling -1.7)
     ⇒ -1
```

`round` *number* **&optional** *divisor* [Function]

This returns *number*, converted to an integer by rounding towards the nearest integer. Rounding a value equidistant between two integers returns the even integer.

```
(round 1.2)
     ⇒ 1
(round 1.7)
     ⇒ 2
(round -1.2)
     ⇒ -1
(round -1.7)
     ⇒ -2
```

3.6 Arithmetic Operations

Emacs Lisp provides the traditional four arithmetic operations (addition, subtraction, multiplication, and division), as well as remainder and modulus functions, and functions to add or subtract 1. Except for `%`, each of these functions accepts both integer and floating-point arguments, and returns a floating-point number if any argument is floating point.

`1+` *number-or-marker* [Function]

This function returns *number-or-marker* plus 1. For example,

```
(setq foo 4)
     ⇒ 4
(1+ foo)
     ⇒ 5
```

This function is not analogous to the C operator `++`—it does not increment a variable. It just computes a sum. Thus, if we continue,

```
foo
     ⇒ 4
```

If you want to increment the variable, you must use `setq`, like this:

```
(setq foo (1+ foo))
     ⇒ 5
```

`1-` *number-or-marker* [Function]

This function returns *number-or-marker* minus 1.

`+` **&rest** *numbers-or-markers* [Function]

This function adds its arguments together. When given no arguments, `+` returns 0.

```
(+)
     ⇒ 0
(+ 1)
     ⇒ 1
(+ 1 2 3 4)
     ⇒ 10
```

- **&optional** *number-or-marker* **&rest** *more-numbers-or-markers* [Function]

The - function serves two purposes: negation and subtraction. When - has a single argument, the value is the negative of the argument. When there are multiple arguments, - subtracts each of the *more-numbers-or-markers* from *number-or-marker*, cumulatively. If there are no arguments, the result is 0.

```
(- 10 1 2 3 4)
     ⇒ 0
(- 10)
     ⇒ -10
(-)
     ⇒ 0
```

***** **&rest** *numbers-or-markers* [Function]

This function multiplies its arguments together, and returns the product. When given no arguments, * returns 1.

```
(*)
     ⇒ 1
(* 1)
     ⇒ 1
(* 1 2 3 4)
     ⇒ 24
```

/ *number* **&rest** *divisors* [Function]

With one or more *divisors*, this function divides *number* by each divisor in *divisors* in turn, and returns the quotient. With no *divisors*, this function returns 1/*number*, i.e., the multiplicative inverse of *number*. Each argument may be a number or a marker.

If all the arguments are integers, the result is an integer, obtained by rounding the quotient towards zero after each division.

```
(/ 6 2)
     ⇒ 3
(/ 5 2)
     ⇒ 2
(/ 5.0 2)
     ⇒ 2.5
(/ 5 2.0)
     ⇒ 2.5
(/ 5.0 2.0)
     ⇒ 2.5
(/ 4.0)
     ⇒ 0.25
(/ 4)
     ⇒ 0
(/ 25 3 2)
     ⇒ 4
(/ -17 6)
     ⇒ -2
```

If you divide an integer by the integer 0, Emacs signals an `arith-error` error (see Section 11.7.3 [Errors], page 179). On systems using IEEE-754 floating-point, floating-point division of a nonzero number by zero yields either positive or negative infinity (see Section 3.2 [Float Basics], page 39); otherwise, an `arith-error` is signaled as usual.

`%` *dividend divisor* [Function]

This function returns the integer remainder after division of *dividend* by *divisor*. The arguments must be integers or markers.

For any two integers *dividend* and *divisor*,

```
(+ (% dividend divisor)
   (* (/ dividend divisor) divisor))
```

always equals *dividend* if *divisor* is nonzero.

```
(% 9 4)
     ⇒ 1
(% -9 4)
     ⇒ -1
(% 9 -4)
     ⇒ 1
(% -9 -4)
     ⇒ -1
```

`mod` *dividend divisor* [Function]

This function returns the value of *dividend* modulo *divisor*; in other words, the remainder after division of *dividend* by *divisor*, but with the same sign as *divisor*. The arguments must be numbers or markers.

Unlike `%`, `mod` permits floating-point arguments; it rounds the quotient downward (towards minus infinity) to an integer, and uses that quotient to compute the remainder.

If *divisor* is zero, `mod` signals an `arith-error` error if both arguments are integers, and returns a NaN otherwise.

```
(mod 9 4)
     ⇒ 1
(mod -9 4)
     ⇒ 3
(mod 9 -4)
     ⇒ -3
(mod -9 -4)
     ⇒ -1
(mod 5.5 2.5)
     ⇒ .5
```

For any two numbers *dividend* and *divisor*,

```
(+ (mod dividend divisor)
   (* (floor dividend divisor) divisor))
```

always equals *dividend*, subject to rounding error if either argument is floating point and to an `arith-error` if *dividend* is an integer and *divisor* is 0. For `floor`, see Section 3.5 [Numeric Conversions], page 43.

3.7 Rounding Operations

The functions `ffloor`, `fceiling`, `fround`, and `ftruncate` take a floating-point argument and return a floating-point result whose value is a nearby integer. `ffloor` returns the nearest integer below; `fceiling`, the nearest integer above; `ftruncate`, the nearest integer in the direction towards zero; `fround`, the nearest integer.

`ffloor` *float* [Function]
: This function rounds *float* to the next lower integral value, and returns that value as a floating-point number.

`fceiling` *float* [Function]
: This function rounds *float* to the next higher integral value, and returns that value as a floating-point number.

`ftruncate` *float* [Function]
: This function rounds *float* towards zero to an integral value, and returns that value as a floating-point number.

`fround` *float* [Function]
: This function rounds *float* to the nearest integral value, and returns that value as a floating-point number. Rounding a value equidistant between two integers returns the even integer.

3.8 Bitwise Operations on Integers

In a computer, an integer is represented as a binary number, a sequence of *bits* (digits which are either zero or one). Conceptually the bit sequence is infinite on the left, with the most-significant bits being all zeros or all ones. A bitwise operation acts on the individual bits of such a sequence. For example, *shifting* moves the whole sequence left or right one or more places, reproducing the same pattern moved over.

The bitwise operations in Emacs Lisp apply only to integers.

`ash` *integer1 count* [Function]
: `ash` (*arithmetic shift*) shifts the bits in *integer1* to the left *count* places, or to the right if *count* is negative. Left shifts introduce zero bits on the right; right shifts discard the rightmost bits. Considered as an integer operation, `ash` multiplies *integer1* by 2^{count}, and then converts the result to an integer by rounding downward, toward minus infinity.

  Here are examples of `ash`, shifting a pattern of bits one place to the left and to the right. These examples show only the low-order bits of the binary pattern; leading bits all agree with the highest-order bit shown. As you can see, shifting left by one is equivalent to multiplying by two, whereas shifting right by one is equivalent to dividing by two and then rounding toward minus infinity.

```
(ash 7 1) ⇒ 14
;; Decimal 7 becomes decimal 14.
...000111
     ⇒
...001110
```

```
(ash 7 -1) ⇒ 3
...000111
     ⇒
...000011

(ash -7 1) ⇒ -14
...111001
     ⇒
...110010

(ash -7 -1) ⇒ -4
...111001
     ⇒
...111100
```

Here are examples of shifting left or right by two bits:

```
                    ;        binary values
(ash 5 2)           ;   5  =  ...000101
     ⇒ 20           ;      =  ...010100
(ash -5 2)          ;  -5  =  ...111011
     ⇒ -20          ;      =  ...101100
(ash 5 -2)
     ⇒ 1            ;      =  ...000001
(ash -5 -2)
     ⇒ -2           ;      =  ...111110
```

`lsh` *integer1 count* [Function]

`lsh`, which is an abbreviation for *logical shift*, shifts the bits in *integer1* to the left *count* places, or to the right if *count* is negative, bringing zeros into the vacated bits. If *count* is negative, then *integer1* must be either a fixnum or a positive bignum, and `lsh` treats a negative fixnum as if it were unsigned by subtracting twice `most-negative-fixnum` before shifting, producing a nonnegative result. This quirky behavior dates back to when Emacs supported only fixnums; nowadays `ash` is a better choice.

As `lsh` behaves like `ash` except when *integer1* and *count1* are both negative, the following examples focus on these exceptional cases. These examples assume 30-bit fixnums.

```
                    ;     binary values
(ash -7 -1)         ; -7 = ...11111111111111111111111111001
     ⇒ -4           ;    = ...11111111111111111111111111100
(lsh -7 -1)
     ⇒ 536870908 ;    = ...01111111111111111111111111100
(ash -5 -2)         ; -5 = ...11111111111111111111111111011
     ⇒ -2           ;    = ...11111111111111111111111111110
(lsh -5 -2)
     ⇒ 268435454 ;    = ...00111111111111111111111111110
```

`logand` **&rest** *ints-or-markers* [Function]

This function returns the bitwise AND of the arguments: the *n*th bit is 1 in the result if, and only if, the *n*th bit is 1 in all the arguments.

For example, using 4-bit binary numbers, the bitwise AND of 13 and 12 is 12: 1101 combined with 1100 produces 1100. In both the binary numbers, the leftmost two

bits are both 1 so the leftmost two bits of the returned value are both 1. However, for the rightmost two bits, each is 0 in at least one of the arguments, so the rightmost two bits of the returned value are both 0.

Therefore,

```
(logand 13 12)
     ⇒ 12
```

If `logand` is not passed any argument, it returns a value of −1. This number is an identity element for `logand` because its binary representation consists entirely of ones. If `logand` is passed just one argument, it returns that argument.

```
                        ;       binary values

(logand 14 13)          ; 14  =  ...001110
                        ; 13  =  ...001101
     ⇒ 12               ; 12  =  ...001100

(logand 14 13 4)        ; 14  =  ...001110
                        ; 13  =  ...001101
                        ;  4  =  ...000100
     ⇒ 4                ;  4  =  ...000100

(logand)
     ⇒ -1               ; -1  =  ...111111
```

`logior` **&rest** *ints-or-markers* [Function]

This function returns the bitwise inclusive OR of its arguments: the *n*th bit is 1 in the result if, and only if, the *n*th bit is 1 in at least one of the arguments. If there are no arguments, the result is 0, which is an identity element for this operation. If `logior` is passed just one argument, it returns that argument.

```
                        ;       binary values

(logior 12 5)           ; 12  =  ...001100
                        ;  5  =  ...000101
     ⇒ 13               ; 13  =  ...001101

(logior 12 5 7)         ; 12  =  ...001100
                        ;  5  =  ...000101
                        ;  7  =  ...000111
     ⇒ 15               ; 15  =  ...001111
```

`logxor` **&rest** *ints-or-markers* [Function]

This function returns the bitwise exclusive OR of its arguments: the *n*th bit is 1 in the result if, and only if, the *n*th bit is 1 in an odd number of the arguments. If there are no arguments, the result is 0, which is an identity element for this operation. If `logxor` is passed just one argument, it returns that argument.

```
                        ;       binary values

(logxor 12 5)           ; 12  =  ...001100
                        ;  5  =  ...000101
     ⇒ 9                ;  9  =  ...001001

(logxor 12 5 7)         ; 12  =  ...001100
                        ;  5  =  ...000101
                        ;  7  =  ...000111
     ⇒ 14               ; 14  =  ...001110
```

`lognot` *integer* [Function]
: This function returns the bitwise complement of its argument: the *n*th bit is one in the result if, and only if, the *n*th bit is zero in *integer*, and vice-versa. The result equals $-1 - integer$.

```
(lognot 5)
     ⇒ -6
;;  5  =  ...000101
;; becomes
;; -6  =  ...111010
```

`logcount` *integer* [Function]
: This function returns the *Hamming weight* of *integer*: the number of ones in the binary representation of *integer*. If *integer* is negative, it returns the number of zero bits in its two's complement binary representation. The result is always nonnegative.

```
(logcount 43)      ;  43 = ...000101011
     ⇒ 4
(logcount -43)     ; -43 = ...111010101
     ⇒ 3
```

3.9 Standard Mathematical Functions

These mathematical functions allow integers as well as floating-point numbers as arguments.

`sin` *arg* [Function]
`cos` *arg* [Function]
`tan` *arg* [Function]
: These are the basic trigonometric functions, with argument *arg* measured in radians.

`asin` *arg* [Function]
: The value of `(asin arg)` is a number between $-\pi/2$ and $\pi/2$ (inclusive) whose sine is *arg*. If *arg* is out of range (outside $[-1, 1]$), `asin` returns a NaN.

`acos` *arg* [Function]
: The value of `(acos arg)` is a number between 0 and π (inclusive) whose cosine is *arg*. If *arg* is out of range (outside $[-1, 1]$), `acos` returns a NaN.

`atan` *y* **&optional** *x* [Function]
: The value of `(atan y)` is a number between $-\pi/2$ and $\pi/2$ (exclusive) whose tangent is *y*. If the optional second argument *x* is given, the value of `(atan y x)` is the angle in radians between the vector `[x, y]` and the `X` axis.

`exp` *arg* [Function]
: This is the exponential function; it returns e to the power *arg*.

`log` *arg* **&optional** *base* [Function]
: This function returns the logarithm of *arg*, with base *base*. If you don't specify *base*, the natural base e is used. If *arg* or *base* is negative, `log` returns a NaN.

`expt` *x y* [Function]
: This function returns *x* raised to power *y*. If both arguments are integers and *y* is nonnegative, the result is an integer; in this case, overflow signals an error, so watch out. If *x* is a finite negative number and *y* is a finite non-integer, `expt` returns a NaN.

`sqrt` *arg* [Function]
: This returns the square root of *arg*. If *arg* is finite and less than zero, `sqrt` returns a NaN.

In addition, Emacs defines the following common mathematical constants:

`float-e` [Variable]
: The mathematical constant *e* (2.71828...).

`float-pi` [Variable]
: The mathematical constant *pi* (3.14159...).

3.10 Random Numbers

A deterministic computer program cannot generate true random numbers. For most purposes, *pseudo-random numbers* suffice. A series of pseudo-random numbers is generated in a deterministic fashion. The numbers are not truly random, but they have certain properties that mimic a random series. For example, all possible values occur equally often in a pseudo-random series.

Pseudo-random numbers are generated from a *seed value*. Starting from any given seed, the `random` function always generates the same sequence of numbers. By default, Emacs initializes the random seed at startup, in such a way that the sequence of values of `random` (with overwhelming likelihood) differs in each Emacs run. The random seed is typically initialized from system entropy; however, on obsolescent platforms lacking entropy pools, the seed is taken from less-random volatile data such as the current time.

Sometimes you want the random number sequence to be repeatable. For example, when debugging a program whose behavior depends on the random number sequence, it is helpful to get the same behavior in each program run. To make the sequence repeat, execute `(random "")`. This sets the seed to a constant value for your particular Emacs executable (though it may differ for other Emacs builds). You can use other strings to choose various seed values.

`random` **&optional** *limit* [Function]
: This function returns a pseudo-random integer. Repeated calls return a series of pseudo-random integers.

  If *limit* is a positive integer, the value is chosen to be nonnegative and less than *limit*. Otherwise, the value might be any fixnum, i.e., any integer from `most-negative-fixnum` through `most-positive-fixnum` (see Section 3.1 [Integer Basics], page 38).

  If *limit* is a string, it means to choose a new seed based on the string's contents. This causes later calls to `random` to return a reproducible sequence of results.

  If *limit* is `t`, it means to choose a new seed as if Emacs were restarting. This causes later calls to `random` to return an unpredictable sequence of results.

If you need a random nonce for cryptographic purposes, using `random` is typically not the best approach, for several reasons:

- Although you can use `(random t)` to consult system entropy, doing so can adversely affect other parts of your program that benefit from reproducible results.
- The system-dependent pseudo-random number generator (PRNG) used by `random` is not necessarily suitable for cryptography.
- A call to `(random t)` does not give direct access to system entropy; the entropy is passed through the system-dependent PRNG, thus possibly biasing the results.
- On typical platforms the random seed contains only 32 bits, which is typically narrower than an Emacs fixnum, and is not nearly enough for cryptographic purposes.
- A `(random t)` call leaves information about the nonce scattered about Emacs's internal state, increasing the size of the internal attack surface.
- On obsolescent platforms lacking entropy pools, `(random t)` is seeded from a cryptographically weak source.

4 Strings and Characters

A string in Emacs Lisp is an array that contains an ordered sequence of characters. Strings are used as names of symbols, buffers, and files; to send messages to users; to hold text being copied between buffers; and for many other purposes. Because strings are so important, Emacs Lisp has many functions expressly for manipulating them. Emacs Lisp programs use strings more often than individual characters.

See Section 22.7.17 [Strings of Events], page 460, for special considerations for strings of keyboard character events.

4.1 String and Character Basics

A character is a Lisp object which represents a single character of text. In Emacs Lisp, characters are simply integers; whether an integer is a character or not is determined only by how it is used. See Section 34.5 [Character Codes], page 970, for details about character representation in Emacs.

A string is a fixed sequence of characters. It is a type of sequence called an *array*, meaning that its length is fixed and cannot be altered once it is created (see Chapter 6 [Sequences Arrays Vectors], page 102). Unlike in C, Emacs Lisp strings are *not* terminated by a distinguished character code.

Since strings are arrays, and therefore sequences as well, you can operate on them with the general array and sequence functions documented in Chapter 6 [Sequences Arrays Vectors], page 102. For example, you can access individual characters in a string using the function `aref` (see Section 6.3 [Array Functions], page 116).

There are two text representations for non-ASCII characters in Emacs strings (and in buffers): unibyte and multibyte. For most Lisp programming, you don't need to be concerned with these two representations. See Section 34.1 [Text Representations], page 966, for details.

Sometimes key sequences are represented as unibyte strings. When a unibyte string is a key sequence, string elements in the range 128 to 255 represent meta characters (which are large integers) rather than character codes in the range 128 to 255. Strings cannot hold characters that have the hyper, super or alt modifiers; they can hold ASCII control characters, but no other control characters. They do not distinguish case in ASCII control characters. If you want to store such characters in a sequence, such as a key sequence, you must use a vector instead of a string. See Section 2.4.3 [Character Type], page 11, for more information about keyboard input characters.

Strings are useful for holding regular expressions. You can also match regular expressions against strings with `string-match` (see Section 35.4 [Regexp Search], page 1011). The functions `match-string` (see Section 35.6.2 [Simple Match Data], page 1016) and `replace-match` (see Section 35.6.1 [Replacing Match], page 1015) are useful for decomposing and modifying strings after matching regular expressions against them.

Like a buffer, a string can contain text properties for the characters in it, as well as the characters themselves. See Section 33.19 [Text Properties], page 919. All the Lisp primitives that copy text from strings to buffers or other strings also copy the properties of the characters being copied.

See Chapter 33 [Text], page 880, for information about functions that display strings or copy them into buffers. See Section 2.4.3 [Character Type], page 11, and Section 2.4.8 [String Type], page 20, for information about the syntax of characters and strings. See Chapter 34 [Non-ASCII Characters], page 966, for functions to convert between text representations and to encode and decode character codes. Also, note that `length` should *not* be used for computing the width of a string on display; use `string-width` (see Section 41.10 [Size of Displayed Text], page 1161) instead.

4.2 Predicates for Strings

For more information about general sequence and array predicates, see Chapter 6 [Sequences Arrays Vectors], page 102, and Section 6.2 [Arrays], page 115.

`stringp` *object* [Function]
: This function returns `t` if *object* is a string, `nil` otherwise.

`string-or-null-p` *object* [Function]
: This function returns `t` if *object* is a string or `nil`. It returns `nil` otherwise.

`char-or-string-p` *object* [Function]
: This function returns `t` if *object* is a string or a character (i.e., an integer), `nil` otherwise.

4.3 Creating Strings

The following functions create strings, either from scratch, or by putting strings together, or by taking them apart. (For functions that create strings based on the modified contents of other strings, like `string-replace` and `replace-regexp-in-string`, see Section 35.7 [Search and Replace], page 1019.)

`make-string` *count character* **&optional** *multibyte* [Function]
: This function returns a string made up of *count* repetitions of *character*. If *count* is negative, an error is signaled.

```
(make-string 5 ?x)
     ⇒ "xxxxx"
(make-string 0 ?x)
     ⇒ ""
```

Normally, if *character* is an ASCII character, the result is a unibyte string. But if the optional argument *multibyte* is non-`nil`, the function will produce a multibyte string instead. This is useful when you later need to concatenate the result with non-ASCII strings or replace some of its characters with non-ASCII characters.

Other functions to compare with this one include `make-vector` (see Section 6.4 [Vectors], page 117) and `make-list` (see Section 5.4 [Building Lists], page 82).

`string` **&rest** *characters* [Function]
: This returns a string containing the characters *characters*.

```
(string ?a ?b ?c)
     ⇒ "abc"
```

`substring` *string* **&optional** *start end* [Function]

This function returns a new string which consists of those characters from *string* in the range from (and including) the character at the index *start* up to (but excluding) the character at the index *end*. The first character is at index zero. With one argument, this function just copies *string*.

```
(substring "abcdefg" 0 3)
     ⇒ "abc"
```

In the above example, the index for '`a`' is 0, the index for '`b`' is 1, and the index for '`c`' is 2. The index 3—which is the fourth character in the string—marks the character position up to which the substring is copied. Thus, '`abc`' is copied from the string `"abcdefg"`.

A negative number counts from the end of the string, so that −1 signifies the index of the last character of the string. For example:

```
(substring "abcdefg" -3 -1)
     ⇒ "ef"
```

In this example, the index for '`e`' is −3, the index for '`f`' is −2, and the index for '`g`' is −1. Therefore, '`e`' and '`f`' are included, and '`g`' is excluded.

When `nil` is used for *end*, it stands for the length of the string. Thus,

```
(substring "abcdefg" -3 nil)
     ⇒ "efg"
```

Omitting the argument *end* is equivalent to specifying `nil`. It follows that `(substring` *`string`* `0)` returns a copy of all of *string*.

```
(substring "abcdefg" 0)
     ⇒ "abcdefg"
```

But we recommend `copy-sequence` for this purpose (see Section 6.1 [Sequence Functions], page 102).

If the characters copied from *string* have text properties, the properties are copied into the new string also. See Section 33.19 [Text Properties], page 919.

`substring` also accepts a vector for the first argument. For example:

```
(substring [a b (c) "d"] 1 3)
     ⇒ [b (c)]
```

A `wrong-type-argument` error is signaled if *start* is not an integer or if *end* is neither an integer nor `nil`. An `args-out-of-range` error is signaled if *start* indicates a character following *end*, or if either integer is out of range for *string*.

Contrast this function with `buffer-substring` (see Section 33.2 [Buffer Contents], page 881), which returns a string containing a portion of the text in the current buffer. The beginning of a string is at index 0, but the beginning of a buffer is at index 1.

`substring-no-properties` *string* **&optional** *start end* [Function]

This works like `substring` but discards all text properties from the value. Also, *start* may be omitted or `nil`, which is equivalent to 0. Thus, `(substring-no-properties` *`string`*`)` returns a copy of *string*, with all text properties removed.

`concat` **&rest** *sequences* [Function]

This function returns a string consisting of the characters in the arguments passed to it (along with their text properties, if any). The arguments may be strings, lists of numbers, or vectors of numbers; they are not themselves changed. If `concat` receives no arguments, it returns an empty string.

```
(concat "abc" "-def")
     ⇒ "abc-def"
(concat "abc" (list 120 121) [122])
     ⇒ "abcxyz"
;; nil is an empty sequence.
(concat "abc" nil "-def")
     ⇒ "abc-def"
(concat "The " "quick brown " "fox.")
     ⇒ "The quick brown fox."
(concat)
     ⇒ ""
```

This function does not always allocate a new string. Callers are advised not rely on the result being a new string nor on it being `eq` to an existing string.

In particular, mutating the returned value may inadvertently change another string, alter a constant string in the program, or even raise an error. To obtain a string that you can safely mutate, use `copy-sequence` on the result.

For information about other concatenation functions, see the description of `mapconcat` in Section 13.6 [Mapping Functions], page 241, `vconcat` in Section 6.5 [Vector Functions], page 118, and `append` in Section 5.4 [Building Lists], page 82. For concatenating individual command-line arguments into a string to be used as a shell command, see Section 40.2 [Shell Arguments], page 1083.

`split-string` *string* **&optional** *separators omit-nulls trim* [Function]

This function splits *string* into substrings based on the regular expression *separators* (see Section 35.3 [Regular Expressions], page 999). Each match for *separators* defines a splitting point; the substrings between splitting points are made into a list, which is returned.

If *separators* is `nil` (or omitted), the default is the value of `split-string-default-separators` and the function behaves as if *omit-nulls* were `t`.

If *omit-nulls* is `nil` (or omitted), the result contains null strings whenever there are two consecutive matches for *separators*, or a match is adjacent to the beginning or end of *string*. If *omit-nulls* is `t`, these null strings are omitted from the result.

If the optional argument *trim* is non-`nil`, it should be a regular expression to match text to trim from the beginning and end of each substring. If trimming makes the substring empty, it is treated as null.

If you need to split a string into a list of individual command-line arguments suitable for `call-process` or `start-process`, see Section 40.2 [Shell Arguments], page 1083.

Examples:

```
(split-string "  two words ")
     ⇒ ("two" "words")
```

The result is not `("" "two" "words" "")`, which would rarely be useful. If you need such a result, use an explicit value for *separators*:

```
(split-string "  two words "
              split-string-default-separators)
     ⇒ ("" "two" "words" "")

(split-string "Soup is good food" "o")
     ⇒ ("S" "up is g" "" "d f" "" "d")
(split-string "Soup is good food" "o" t)
     ⇒ ("S" "up is g" "d f" "d")
(split-string "Soup is good food" "o+")
     ⇒ ("S" "up is g" "d f" "d")
```

Empty matches do count, except that `split-string` will not look for a final empty match when it already reached the end of the string using a non-empty match or when *string* is empty:

```
(split-string "aooob" "o*")
     ⇒ ("" "a" "" "b" "")
(split-string "ooaboo" "o*")
     ⇒ ("" "" "a" "b" "")
(split-string "" "")
     ⇒ ("")
```

However, when *separators* can match the empty string, *omit-nulls* is usually `t`, so that the subtleties in the three previous examples are rarely relevant:

```
(split-string "Soup is good food" "o*" t)
     ⇒ ("S" "u" "p" " " "i" "s" " " "g" "d" " " "f" "d")
(split-string "Nice doggy!" "" t)
     ⇒ ("N" "i" "c" "e" " " "d" "o" "g" "g" "y" "!")
(split-string "" "" t)
     ⇒ nil
```

Somewhat odd, but predictable, behavior can occur for certain "non-greedy" values of *separators* that can prefer empty matches over non-empty matches. Again, such values rarely occur in practice:

```
(split-string "ooo" "o*" t)
     ⇒ nil
(split-string "ooo" "\\|o+" t)
     ⇒ ("o" "o" "o")
```

`split-string-default-separators` [Variable]

The default value of *separators* for `split-string`. Its usual value is `"[ \f\t\n\r\v]+"`.

`string-clean-whitespace` *string* [Function]

Clean up the whitespace in *string* by collapsing stretches of whitespace to a single space character, as well as removing all whitespace from the start and the end of *string*.

`string-trim-left` *string* **&optional** *regexp* [Function]
Remove the leading text that matches *regexp* from *string*. *regexp* defaults to '`[ \t\n\r]+`'.

`string-trim-right` *string* **&optional** *regexp* [Function]
Remove the trailing text that matches *regexp* from *string*. *regexp* defaults to '`[ \t\n\r]+`'.

`string-trim` *string* **&optional** *trim-left trim-right* [Function]
Remove the leading text that matches *trim-left* and trailing text that matches *trim-right* from *string*. Both regexps default to '`[ \t\n\r]+`'.

`string-fill` *string length* [Function]
Attempt to Word-wrap *string* so that no lines are longer than *length*. Filling is done on whitespace boundaries only. If there are individual words that are longer than *length*, these will not be shortened.

`string-limit` *string length* **&optional** *end coding-system* [Function]
If *string* is shorter than *length* characters, *string* is returned as is. Otherwise, return a substring of *string* consisting of the first *length* characters. If the optional *end* parameter is given, return a string of the *length* last characters instead.

If *coding-system* is non-`nil`, *string* will be encoded before limiting, and the result will be a unibyte string that's shorter than `length` bytes. If *string* contains characters that are encoded into several bytes (for instance, when using `utf-8`), the resulting unibyte string is never truncated in the middle of a character representation.

This function measures the string length in characters or bytes, and thus is generally inappropriate if you need to shorten strings for display purposes; use `truncate-string-to-width` or `window-text-pixel-size` or `string-glyph-split` instead (see Section 41.10 [Size of Displayed Text], page 1161).

`string-lines` *string* **&optional** *omit-nulls keep-newlines* [Function]
Split *string* into a list of strings on newline boundaries. If the optional argument *omit-nulls* is non-`nil`, remove empty lines from the results. If the optional argument *keep-newlines* is non-`nil`, don't remove the trailing newlines from the result strings.

`string-pad` *string length* **&optional** *padding start* [Function]
Pad *string* to be of the given *length* using *padding* as the padding character. *padding* defaults to the space character. If *string* is longer than *length*, no padding is done. If *start* is `nil` or omitted, the padding is appended to the characters of *string*, and if it's non-`nil`, the padding is prepended to *string*'s characters.

`string-chop-newline` *string* [Function]
Remove the final newline, if any, from *string*.

4.4 Modifying Strings

You can alter the contents of a mutable string via operations described in this section. See Section 2.9 [Mutability], page 37.

The most basic way to alter the contents of an existing string is with `aset` (see Section 6.3 [Array Functions], page 116). `(aset string idx char)` stores *char* into *string* at character index *idx*. It will automatically convert a pure-ASCII *string* to a multibyte string (see Section 34.1 [Text Representations], page 966) if needed, but we recommend to always make sure *string* is multibyte (e.g., by using `string-to-multibyte`, see Section 34.3 [Converting Representations], page 968), if *char* is a non-ASCII character, not a raw byte.

A more powerful function is `store-substring`:

`store-substring` *string idx obj* [Function]
: This function alters part of the contents of the specified *string*, by storing *obj* starting at character index *idx*. The argument *obj* may be either a character (in which case the function behaves exactly as `aset`) or a (smaller) string. If *obj* is a multibyte string, we recommend to make sure *string* is also multibyte, even if it's pure-ASCII.

  Since it is impossible to change the number of characters in an existing string, it is an error if *obj* consists of more characters than would fit in *string* starting at character index *idx*.

To clear out a string that contained a password, use `clear-string`:

`clear-string` *string* [Function]
: This makes *string* a unibyte string and clears its contents to zeros. It may also change *string*'s length.

4.5 Comparison of Characters and Strings

`char-equal` *character1 character2* [Function]
: This function returns `t` if the arguments represent the same character, `nil` otherwise. This function ignores differences in case if `case-fold-search` is non-`nil`.

```
(char-equal ?x ?x)
     ⇒ t
(let ((case-fold-search nil))
  (char-equal ?x ?X))
     ⇒ nil
```

`string=` *string1 string2* [Function]
: This function returns `t` if the characters of the two strings match exactly. Symbols are also allowed as arguments, in which case the symbol names are used. Case is always significant, regardless of `case-fold-search`.

  This function is equivalent to `equal` for comparing two strings (see Section 2.8 [Equality Predicates], page 34). In particular, the text properties of the two strings are ignored; use `equal-including-properties` if you need to distinguish between strings that differ only in their text properties. However, unlike `equal`, if either argument is not a string or symbol, `string=` signals an error.

```
(string= "abc" "abc")
     ⇒ t
(string= "abc" "ABC")
     ⇒ nil
```

```
(string= "ab" "ABC")
     ⇒ nil
```

A unibyte and a multibyte string are equal in the sense of `string=` if and only if they contain the same sequence of character codes all being in the range 0–127 (ASCII). See Section 34.1 [Text Representations], page 966.

`string-equal` *string1 string2* [Function]

`string-equal` is another name for `string=`.

`string-equal-ignore-case` *string1 string2* [Function]

`string-equal-ignore-case` compares strings ignoring case differences, like `char-equal` when `case-fold-search` is `t`.

`string-collate-equalp` *string1 string2* **&optional** *locale ignore-case* [Function]

This function returns `t` if *string1* and *string2* are equal with respect to the collation rules of the specified *locale*, which defaults to your current system locale. A collation rule is not only determined by the lexicographic order of the characters contained in *string1* and *string2*, but also by further rules about relations between these characters. Usually, it is defined by the locale environment with which Emacs is running and by the Standard C library against which Emacs was linked[1].

For example, characters with different code points but the same meaning, like different grave accent Unicode characters, might, in some locales, be considered as equal:

```
(string-collate-equalp (string ?\uFF40) (string ?\u1FEF))
     ⇒ t
```

The optional argument *locale*, a string, overrides the setting of your current locale identifier for collation. The value is system dependent; a *locale* `"en_US.UTF-8"` is applicable on POSIX systems, while it would be, e.g., `"enu_USA.1252"` on MS-Windows systems.

If *ignore-case* is non-`nil`, characters are compared case-insensitively, by converting them to lower-case. However, if the underlying system library doesn't provide locale-specific collation rules, this function falls back to `string-equal`, in which case the *ignore-case* argument is ignored, and the comparison will always be case-sensitive.

To emulate Unicode-compliant collation on MS-Windows systems, bind `w32-collate-ignore-punctuation` to a non-`nil` value, since the codeset part of the locale cannot be `"UTF-8"` on MS-Windows.

If your system does not support a locale environment, this function behaves like `string-equal`.

Do *not* use this function to compare file names for equality, as filesystems generally don't honor linguistic equivalence of strings that collation implements.

`string<` *string1 string2* [Function]

This function compares two strings a character at a time. It scans both the strings at the same time to find the first pair of corresponding characters that do not match. If

[1] For more information about collation rules and their locale dependencies, see The Unicode Collation Algorithm (`https://unicode.org/reports/tr10/`). Some Standard C libraries, such as the GNU C Library (a.k.a. *glibc*) implement large portions of the Unicode Collation Algorithm and use the associated locale data, Common Locale Data Repository, or CLDR.

the lesser character of these two is the character from *string1*, then *string1* is less, and this function returns `t`. If the lesser character is the one from *string2*, then *string1* is greater, and this function returns `nil`. If the two strings match entirely, the value is `nil`.

Pairs of characters are compared according to their character codes. Keep in mind that lower case letters have higher numeric values in the ASCII character set than their upper case counterparts; digits and many punctuation characters have a lower numeric value than upper case letters. An ASCII character is less than any non-ASCII character; a unibyte non-ASCII character is always less than any multibyte non-ASCII character (see Section 34.1 [Text Representations], page 966).

```
(string< "abc" "abd")
     ⇒ t
(string< "abd" "abc")
     ⇒ nil
(string< "123" "abc")
     ⇒ t
```

When the strings have different lengths, and they match up to the length of *string1*, then the result is `t`. If they match up to the length of *string2*, the result is `nil`. A string of no characters is less than any other string.

```
(string< "" "abc")
     ⇒ t
(string< "ab" "abc")
     ⇒ t
(string< "abc" "")
     ⇒ nil
(string< "abc" "ab")
     ⇒ nil
(string< "" "")
     ⇒ nil
```

Symbols are also allowed as arguments, in which case their print names are compared.

`string-lessp` *string1 string2* [Function]

`string-lessp` is another name for `string<`.

`string-greaterp` *string1 string2* [Function]

This function returns the result of comparing *string1* and *string2* in the opposite order, i.e., it is equivalent to calling `(string-lessp string2 string1)`.

`string-collate-lessp` *string1 string2* **&optional** *locale ignore-case* [Function]

This function returns `t` if *string1* is less than *string2* in collation order of the specified *locale*, which defaults to your current system locale. A collation order is not only determined by the lexicographic order of the characters contained in *string1* and *string2*, but also by further rules about relations between these characters. Usually, it is defined by the locale environment with which Emacs is running, and by the Standard C library against which Emacs was linked.

For example, punctuation and whitespace characters might be ignored for sorting (see Section 6.1 [Sequence Functions], page 102):

```
(sort (list "11" "12" "1 1" "1 2" "1.1" "1.2") 'string-collate-lessp)
     ⇒ ("11" "1 1" "1.1" "12" "1 2" "1.2")
```

This behavior is system-dependent; e.g., punctuation and whitespace are never ignored on Cygwin, regardless of locale.

The optional argument *locale*, a string, overrides the setting of your current locale identifier for collation. The value is system dependent; a *locale* `"en_US.UTF-8"` is applicable on POSIX systems, while it would be, e.g., `"enu_USA.1252"` on MS-Windows systems. The *locale* value of `"POSIX"` or `"C"` lets `string-collate-lessp` behave like `string-lessp`:

```
(sort (list "11" "12" "1 1" "1 2" "1.1" "1.2")
      (lambda (s1 s2) (string-collate-lessp s1 s2 "POSIX")))
     ⇒ ("1 1" "1 2" "1.1" "1.2" "11" "12")
```

If *ignore-case* is non-`nil`, characters are compared case-insensitively, by converting them to lower-case. However, if the underlying system library doesn't provide locale-specific collation rules, this function falls back to `string-lessp`, in which case the *ignore-case* argument is ignored, and the comparison will always be case-sensitive.

To emulate Unicode-compliant collation on MS-Windows systems, bind `w32-collate-ignore-punctuation` to a non-`nil` value, since the codeset part of the locale cannot be `"UTF-8"` on MS-Windows.

If your system does not support a locale environment, this function behaves like `string-lessp`.

`string-version-lessp` *string1 string2* [Function]

This function compares strings lexicographically, except it treats sequences of numerical characters as if they comprised a base-ten number, and then compares the numbers. So '`foo2.png`' is "smaller" than '`foo12.png`' according to this predicate, even if '`12`' is lexicographically "smaller" than '`2`'.

`string-prefix-p` *string1 string2* **&optional** *ignore-case* [Function]

This function returns non-`nil` if *string1* is a prefix of *string2*; i.e., if *string2* starts with *string1*. If the optional argument *ignore-case* is non-`nil`, the comparison ignores case differences.

`string-suffix-p` *suffix string* **&optional** *ignore-case* [Function]

This function returns non-`nil` if *suffix* is a suffix of *string*; i.e., if *string* ends with *suffix*. If the optional argument *ignore-case* is non-`nil`, the comparison ignores case differences.

`string-search` *needle haystack* **&optional** *start-pos* [Function]

Return the position of the first instance of *needle* in *haystack*, both of which are strings. If *start-pos* is non-`nil`, start searching from that position in *haystack*. Return `nil` if no match was found. This function only considers the characters in the strings when doing the comparison; text properties are ignored. Matching is always case-sensitive.

`compare-strings` *string1 start1 end1 string2 start2 end2* **&optional** *ignore-case* [Function]

This function compares a specified part of *string1* with a specified part of *string2*. The specified part of *string1* runs from index *start1* (inclusive) up to index *end1* (exclusive); `nil` for *start1* means the start of the string, while `nil` for *end1* means the length of the string. Likewise, the specified part of *string2* runs from index *start2* up to index *end2*.

The strings are compared by the numeric values of their characters. For instance, *str1* is considered less than *str2* if its first differing character has a smaller numeric value. If *ignore-case* is non-`nil`, characters are converted to upper-case, using the current buffer's case-table (see Section 4.10 [Case Tables], page 74), before comparing them. Unibyte strings are converted to multibyte for comparison (see Section 34.1 [Text Representations], page 966), so that a unibyte string and its conversion to multibyte are always regarded as equal.

If the specified portions of the two strings match, the value is `t`. Otherwise, the value is an integer which indicates how many leading characters agree, and which string is less. Its absolute value is one plus the number of characters that agree at the beginning of the two strings. The sign is negative if *string1* (or its specified portion) is less.

`string-distance` *string1 string2* **&optional** *bytecompare* [Function]

This function returns the *Levenshtein distance* between the source string *string1* and the target string *string2*. The Levenshtein distance is the number of single-character changes—deletions, insertions, or replacements—required to transform the source string into the target string; it is one possible definition of the *edit distance* between strings.

Letter-case of the strings is significant for the computed distance, but their text properties are ignored. If the optional argument *bytecompare* is non-`nil`, the function calculates the distance in terms of bytes instead of characters. The byte-wise comparison uses the internal Emacs representation of characters, so it will produce inaccurate results for multibyte strings that include raw bytes (see Section 34.1 [Text Representations], page 966); make the strings unibyte by encoding them (see Section 34.10.7 [Explicit Encoding], page 991) if you need accurate results with raw bytes.

`assoc-string` *key alist* **&optional** *case-fold* [Function]

This function works like `assoc`, except that *key* must be a string or symbol, and comparison is done using `compare-strings`. Symbols are converted to strings before testing. If *case-fold* is non-`nil`, *key* and the elements of *alist* are converted to upper-case before comparison. Unlike `assoc`, this function can also match elements of the alist that are strings or symbols rather than conses. In particular, *alist* can be a list of strings or symbols rather than an actual alist. See Section 5.8 [Association Lists], page 95.

See also the function `compare-buffer-substrings` in Section 33.3 [Comparing Text], page 884, for a way to compare text in buffers. The function `string-match`, which matches a regular expression against a string, can be used for a kind of string comparison; see Section 35.4 [Regexp Search], page 1011.

4.6 Conversion of Characters and Strings

This section describes functions for converting between characters, strings and integers. `format` (see Section 4.7 [Formatting Strings], page 66) and `prin1-to-string` (see Section 20.5 [Output Functions], page 378) can also convert Lisp objects into strings. `read-from-string` (see Section 20.3 [Input Functions], page 375) can convert a string representation of a Lisp object into an object. The functions `string-to-multibyte` and `string-to-unibyte` convert the text representation of a string (see Section 34.3 [Converting Representations], page 968).

See Chapter 25 [Documentation], page 597, for functions that produce textual descriptions of text characters and general input events (`single-key-description` and `text-char-description`). These are used primarily for making help messages.

`number-to-string` *number* [Function]

This function returns a string consisting of the printed base-ten representation of *number*. The returned value starts with a minus sign if the argument is negative.

```
(number-to-string 256)
     ⇒ "256"
(number-to-string -23)
     ⇒ "-23"
(number-to-string -23.5)
     ⇒ "-23.5"
```

`int-to-string` is a semi-obsolete alias for this function.

See also the function `format` in Section 4.7 [Formatting Strings], page 66.

`string-to-number` *string* **&optional** *base* [Function]

This function returns the numeric value of the characters in *string*. If *base* is non-`nil`, it must be an integer between 2 and 16 (inclusive), and integers are converted in that base. If *base* is `nil`, then base ten is used. Floating-point conversion only works in base ten; we have not implemented other radices for floating-point numbers, because that would be much more work and does not seem useful.

The parsing skips spaces and tabs at the beginning of *string*, then reads as much of *string* as it can interpret as a number in the given base. (On some systems it ignores other whitespace at the beginning, not just spaces and tabs.) If *string* cannot be interpreted as a number, this function returns 0.

```
(string-to-number "256")
     ⇒ 256
(string-to-number "25 is a perfect square.")
     ⇒ 25
(string-to-number "X256")
     ⇒ 0
(string-to-number "-4.5")
     ⇒ -4.5
(string-to-number "1e5")
     ⇒ 100000.0
```

`string-to-int` is an obsolete alias for this function.

`char-to-string` *character* [Function]
: This function returns a new string containing one character, *character*. This function is semi-obsolete because the function `string` is more general. See Section 4.3 [Creating Strings], page 55.

`string-to-char` *string* [Function]
: This function returns the first character in *string*. This is mostly identical to `(aref string 0)`, except that it returns 0 if the string is empty. (The value is also 0 when the first character of *string* is the null character, ASCII code 0.) This function may be eliminated in the future if it does not seem useful enough to retain.

Here are some other functions that can convert to or from a string:

`concat` This function converts a vector or a list into a string. See Section 4.3 [Creating Strings], page 55.

`vconcat` This function converts a string into a vector. See Section 6.5 [Vector Functions], page 118.

`append` This function converts a string into a list. See Section 5.4 [Building Lists], page 82.

`byte-to-string`
: This function converts a byte of character data into a unibyte string. See Section 34.3 [Converting Representations], page 968.

4.7 Formatting Strings

Formatting means constructing a string by substituting computed values at various places in a constant string. This constant string controls how the other values are printed, as well as where they appear; it is called a *format string*.

Formatting is often useful for computing messages to be displayed. In fact, the functions `message` and `error` provide the same formatting feature described here; they differ from `format-message` only in how they use the result of formatting.

`format` *string* **&rest** *objects* [Function]
: This function returns a string equal to *string*, replacing any format specifications with encodings of the corresponding *objects*. The arguments *objects* are the computed values to be formatted.

  The characters in *string*, other than the format specifications, are copied directly into the output, including their text properties, if any. Any text properties of the format specifications are copied to the produced string representations of the argument *objects*.

  The output string need not be newly-allocated. For example, if `x` is the string `"foo"`, the expressions `(eq x (format x))` and `(eq x (format "%s" x))` might both yield `t`.

`format-message` *string* **&rest** *objects* [Function]
: This function acts like `format`, except it also converts any grave accents (‘) and apostrophes (’) in *string* as per the value of `text-quoting-style`.

Typically grave accent and apostrophe in the format translate to matching curved quotes, e.g., `"Missing ‘%s’"` might result in `"Missing ‘foo’"`. See Section 25.4 [Text Quoting Style], page 602, for how to influence or inhibit this translation.

A format specification is a sequence of characters beginning with a ‘`%`’. Thus, if there is a ‘`%d`’ in *string*, the `format` function replaces it with the printed representation of one of the values to be formatted (one of the arguments *objects*). For example:

```
(format "The value of fill-column is %d." fill-column)
     ⇒ "The value of fill-column is 72."
```

Since `format` interprets ‘`%`’ characters as format specifications, you should *never* pass an arbitrary string as the first argument. This is particularly true when the string is generated by some Lisp code. Unless the string is *known* to never include any ‘`%`’ characters, pass `"%s"`, described below, as the first argument, and the string as the second, like this:

```
(format "%s" arbitrary-string)
```

Certain format specifications require values of particular types. If you supply a value that doesn't fit the requirements, an error is signaled.

Here is a table of valid format specifications:

‘`%s`’ Replace the specification with the printed representation of the object, made without quoting (that is, using `princ`, not `prin1`—see Section 20.5 [Output Functions], page 378). Thus, strings are represented by their contents alone, with no ‘`"`’ characters, and symbols appear without ‘`\`’ characters.

If the object is a string, its text properties are copied into the output. The text properties of the ‘`%s`’ itself are also copied, but those of the object take priority.

‘`%S`’ Replace the specification with the printed representation of the object, made with quoting (that is, using `prin1`—see Section 20.5 [Output Functions], page 378). Thus, strings are enclosed in ‘`"`’ characters, and ‘`\`’ characters appear where necessary before special characters.

‘`%o`’ Replace the specification with the base-eight representation of an integer. Negative integers are formatted in a platform-dependent way. The object can also be a floating-point number that is formatted as an integer, dropping any fraction.

‘`%d`’ Replace the specification with the base-ten representation of a signed integer. The object can also be a floating-point number that is formatted as an integer, dropping any fraction.

‘`%x`’
‘`%X`’ Replace the specification with the base-sixteen representation of an integer. Negative integers are formatted in a platform-dependent way. ‘`%x`’ uses lower case and ‘`%X`’ uses upper case. The object can also be a floating-point number that is formatted as an integer, dropping any fraction.

‘`%c`’ Replace the specification with the character which is the value given.

‘`%e`’ Replace the specification with the exponential notation for a floating-point number.

‘`%f`’ Replace the specification with the decimal-point notation for a floating-point number.

‘%g’ Replace the specification with notation for a floating-point number, using either exponential notation or decimal-point notation. The exponential notation is used if the exponent would be less than −4 or greater than or equal to the precision (default: 6). By default, trailing zeros are removed from the fractional portion of the result and a decimal-point character appears only if it is followed by a digit.

‘%%’ Replace the specification with a single ‘%’. This format specification is unusual in that its only form is plain ‘%%’ and that it does not use a value. For example, `(format "%% %d" 30)` returns `"% 30"`.

Any other format character results in an ‘`Invalid format operation`’ error.

Here are several examples, which assume the typical `text-quoting-style` settings:

```
(format "The octal value of %d is %o,
         and the hex value is %x." 18 18 18)
     ⇒ "The octal value of 18 is 22,
         and the hex value is 12."

(format-message
 "The name of this buffer is ‘%s’." (buffer-name))
     ⇒ "The name of this buffer is ‘strings.texi’."

(format-message
 "The buffer object prints as `%s'." (current-buffer))
     ⇒ "The buffer object prints as ‘strings.texi’."
```

By default, format specifications correspond to successive values from *objects*. Thus, the first format specification in *string* uses the first such value, the second format specification uses the second such value, and so on. Any extra format specifications (those for which there are no corresponding values) cause an error. Any extra values to be formatted are ignored.

A format specification can have a *field number*, which is a decimal number immediately after the initial ‘%’, followed by a literal dollar sign ‘$’. It causes the format specification to convert the argument with the given number instead of the next argument. Field numbers start at 1. A format can contain either numbered or unnumbered format specifications but not both, except that ‘%%’ can be mixed with numbered specifications.

```
(format "%2$s, %3$s, %%, %1$s" "x" "y" "z")
     ⇒ "y, z, %, x"
```

After the ‘%’ and any field number, you can put certain *flag characters*.

The flag ‘+’ inserts a plus sign before a nonnegative number, so that it always has a sign. A space character as flag inserts a space before a nonnegative number. (Otherwise, nonnegative numbers start with the first digit.) These flags are useful for ensuring that nonnegative and negative numbers use the same number of columns. They are ignored except for ‘%d’, ‘%e’, ‘%f’, ‘%g’, and if both flags are used, ‘+’ takes precedence.

The flag ‘#’ specifies an alternate form which depends on the format in use. For ‘%o’, it ensures that the result begins with a ‘0’. For ‘%x’ and ‘%X’, it prefixes nonzero results with ‘0x’ or ‘0X’. For ‘%e’ and ‘%f’, the ‘#’ flag means include a decimal point even if the

precision is zero. For '`%g`', it always includes a decimal point, and also forces any trailing zeros after the decimal point to be left in place where they would otherwise be removed.

The flag '`0`' ensures that the padding consists of '`0`' characters instead of spaces. This flag is ignored for non-numerical specification characters like '`%s`', '`%S`' and '`%c`'. These specification characters accept the '`0`' flag, but still pad with *spaces*.

The flag '`-`' causes any padding inserted by the width, if specified, to be inserted on the right rather than the left. If both '`-`' and '`0`' are present, the '`0`' flag is ignored.

```
(format "%06d is padded on the left with zeros" 123)
     ⇒ "000123 is padded on the left with zeros"

(format "'%-6d' is padded on the right" 123)
     ⇒ "'123   ' is padded on the right"

(format "The word '%-7s' actually has %d letters in it."
        "foo" (length "foo"))
     ⇒ "The word 'foo    ' actually has 3 letters in it."
```

A specification can have a *width*, which is a decimal number that appears after any field number and flags. If the printed representation of the object contains fewer characters than this width, `format` extends it with padding. Any padding introduced by the width normally consists of spaces inserted on the left:

```
(format "%5d is padded on the left with spaces" 123)
     ⇒ "  123 is padded on the left with spaces"
```

If the width is too small, `format` does not truncate the object's printed representation. Thus, you can use a width to specify a minimum spacing between columns with no risk of losing information. In the following two examples, '`%7s`' specifies a minimum width of 7. In the first case, the string inserted in place of '`%7s`' has only 3 letters, and needs 4 blank spaces as padding. In the second case, the string `"specification"` is 13 letters wide but is not truncated.

```
(format "The word '%7s' has %d letters in it."
        "foo" (length "foo"))
     ⇒ "The word '    foo' has 3 letters in it."
(format "The word '%7s' has %d letters in it."
        "specification" (length "specification"))
     ⇒ "The word 'specification' has 13 letters in it."
```

All the specification characters allow an optional *precision* after the field number, flags and width, if present. The precision is a decimal-point '`.`' followed by a digit-string. For the floating-point specifications ('`%e`' and '`%f`'), the precision specifies how many digits following the decimal point to show; if zero, the decimal-point itself is also omitted. For '`%g`', the precision specifies how many significant digits to show (significant digits are the first digit before the decimal point and all the digits after it). If the precision of %g is zero or unspecified, it is treated as 1. For '`%s`' and '`%S`', the precision truncates the string to the given width, so '`%.3s`' shows only the first three characters of the representation for *object*. For other specification characters, the effect of precision is what the local library functions of the `printf` family produce.

If you plan to use `read` later on the formatted string to retrieve a copy of the formatted value, use a specification that lets `read` reconstruct the value. To format numbers in this reversible way you can use '`%s`' and '`%S`', to format just integers you can also use '`%d`', and to format just nonnegative integers you can also use '`#x%x`' and '`#o%o`'. Other formats may be problematic; for example, '`%d`' and '`%g`' can mishandle NaNs and can lose precision and type, and '`#x%x`' and '`#o%o`' can mishandle negative integers. See Section 20.3 [Input Functions], page 375.

The functions described in this section accept a fixed set of specification characters. The next section describes a function `format-spec` which can accept custom specification characters, such as '`%a`' or '`%z`'.

4.8 Custom Format Strings

Sometimes it is useful to allow users and Lisp programs alike to control how certain text is generated via custom format control strings. For example, a format string could control how to display someone's forename, surname, and email address. Using the function `format` described in the previous section, the format string could be something like `"%s %s <%s>"`. This approach quickly becomes impractical, however, as it can be unclear which specification character corresponds to which piece of information.

A more convenient format string for such cases would be something like `"%f %l <%e>"`, where each specification character carries more semantic information and can easily be rearranged relative to other specification characters, making such format strings more easily customizable by the user.

The function `format-spec` described in this section performs a similar function to `format`, except it operates on format control strings that use arbitrary specification characters.

`format-spec` *template spec-alist* **&optional** *ignore-missing split* [Function]

This function returns a string produced from the format string *template* according to conversions specified in *spec-alist*, which is an alist (see Section 5.8 [Association Lists], page 95) of the form `(letter . replacement)`. Each specification `%letter` in *template* will be replaced by *replacement* when formatting the resulting string.

The characters in *template*, other than the format specifications, are copied directly into the output, including their text properties, if any. Any text properties of the format specifications are copied to their replacements.

Using an alist to specify conversions gives rise to some useful properties:

- If *spec-alist* contains more unique *letter* keys than there are unique specification characters in *template*, the unused keys are simply ignored.
- If *spec-alist* contains more than one association with the same *letter*, the closest one to the start of the list is used.
- If *template* contains the same specification character more than once, then the same *replacement* found in *spec-alist* is used as a basis for all of that character's substitutions.
- The order of specifications in *template* need not correspond to the order of associations in *spec-alist*.

REPLACEMENT can also be a function taking no arguments, and returning a string to be used for the replacement. It will only be called when the corresponding LETTER is used in the TEMPLATE. This is useful, for example, to avoid prompting for input unless it is needed.

The optional argument *ignore-missing* indicates how to handle specification characters in *template* that are not found in *spec-alist*. If it is `nil` or omitted, the function signals an error; if it is `ignore`, those format specifications are left verbatim in the output, including their text properties, if any; if it is `delete`, those format specifications are removed from the output; any other non-`nil` value is handled like `ignore`, but any occurrences of '`%%`' are also left verbatim in the output.

If the optional argument *split* is non-`nil`, instead of returning a single string, `format-spec` will split the result into a list of strings, based on where the substitutions were performed. For instance:

```
(format-spec "foo %b bar" '((?b . "zot")) nil t)
     ⇒ ("foo " "zot" " bar")
```

The syntax of format specifications accepted by `format-spec` is similar, but not identical, to that accepted by `format`. In both cases, a format specification is a sequence of characters beginning with '`%`' and ending with an alphabetic letter such as '`s`'.

Unlike `format`, which assigns specific meanings to a fixed set of specification characters, `format-spec` accepts arbitrary specification characters and treats them all equally. For example:

```
(setq my-site-info
      (list (cons ?s system-name)
            (cons ?t (symbol-name system-type))
            (cons ?c system-configuration)
            (cons ?v emacs-version)
            (cons ?e invocation-name)
            (cons ?p (number-to-string (emacs-pid)))
            (cons ?a user-mail-address)
            (cons ?n user-full-name)))

(format-spec "%e %v (%c)" my-site-info)
     ⇒ "emacs 27.1 (x86_64-pc-linux-gnu)"

(format-spec "%n <%a>" my-site-info)
     ⇒ "Emacs Developers <emacs-devel@gnu.org>"
```

A format specification can include any number of the following flag characters immediately after the '`%`' to modify aspects of the substitution.

'`0`' This flag causes any padding specified by the width to consist of '`0`' characters instead of spaces.

'`-`' This flag causes any padding specified by the width to be inserted on the right rather than the left.

'`<`' This flag causes the substitution to be truncated on the left to the given width and precision, if specified.

‘>’ This flag causes the substitution to be truncated on the right to the given width and precision, if specified.

‘^’ This flag converts the substituted text to upper case (see Section 4.9 [Case Conversion], page 73).

‘_ (underscore)’
This flag converts the substituted text to lower case (see Section 4.9 [Case Conversion], page 73).

The result of using contradictory flags (for instance, both upper and lower case) is undefined.

As is the case with `format`, a format specification can include a width, which is a decimal number that appears after any flags, and a precision, which is a decimal-point ‘.’ followed by a decimal number that appears after any flags and width.

If a substitution contains fewer characters than its specified width, it is padded on the left:

```
(format-spec "%8a is padded on the left with spaces"
             '((?a . "alpha")))
     ⇒ "   alpha is padded on the left with spaces"
```

If a substitution contains more characters than its specified precision, it is truncated on the right:

```
(format-spec "%.2a is truncated on the right"
             '((?a . "alpha")))
     ⇒ "al is truncated on the right"
```

Here is a more complicated example that combines several aforementioned features:

```
(setq my-battery-info
      (list (cons ?p "73")      ; Percentage
            (cons ?L "Battery") ; Status
            (cons ?t "2:23")    ; Remaining time
            (cons ?c "24330")   ; Capacity
            (cons ?r "10.6")))  ; Rate of discharge

(format-spec "%>^-3L : %3p%% (%05t left)" my-battery-info)
     ⇒ "BAT :  73% (02:23 left)"

(format-spec "%>^-3L : %3p%% (%05t left)"
             (cons (cons ?L "AC")
                   my-battery-info))
     ⇒ "AC  :  73% (02:23 left)"
```

As the examples in this section illustrate, `format-spec` is often used for selectively formatting an assortment of different pieces of information. This is useful in programs that provide user-customizable format strings, as the user can choose to format with a regular syntax and in any desired order only a subset of the information that the program makes available.

4.9 Case Conversion in Lisp

The character case functions change the case of single characters or of the contents of strings. The functions normally convert only alphabetic characters (the letters 'A' through 'Z' and 'a' through 'z', as well as non-ASCII letters); other characters are not altered. You can specify a different case conversion mapping by specifying a case table (see Section 4.10 [Case Tables], page 74).

These functions do not modify the strings that are passed to them as arguments.

The examples below use the characters 'X' and 'x' which have ASCII codes 88 and 120 respectively.

`downcase` *string-or-char* [Function]

This function converts *string-or-char*, which should be either a character or a string, to lower case.

When *string-or-char* is a string, this function returns a new string in which each letter in the argument that is upper case is converted to lower case. When *string-or-char* is a character, this function returns the corresponding lower case character (an integer); if the original character is lower case, or is not a letter, the return value is equal to the original character.

```
(downcase "The cat in the hat")
     ⇒ "the cat in the hat"

(downcase ?X)
     ⇒ 120
```

`upcase` *string-or-char* [Function]

This function converts *string-or-char*, which should be either a character or a string, to upper case.

When *string-or-char* is a string, this function returns a new string in which each letter in the argument that is lower case is converted to upper case. When *string-or-char* is a character, this function returns the corresponding upper case character (an integer); if the original character is upper case, or is not a letter, the return value is equal to the original character.

```
(upcase "The cat in the hat")
     ⇒ "THE CAT IN THE HAT"

(upcase ?x)
     ⇒ 88
```

`capitalize` *string-or-char* [Function]

This function capitalizes strings or characters. If *string-or-char* is a string, the function returns a new string whose contents are a copy of *string-or-char* in which each word has been capitalized. This means that the first character of each word is converted to upper case, and the rest are converted to lower case.

The definition of a word is any sequence of consecutive characters that are assigned to the word constituent syntax class in the current syntax table (see Section 36.2.1 [Syntax Class Table], page 1026).

When *string-or-char* is a character, this function does the same thing as `upcase`.

```
(capitalize "The cat in the hat")
     ⇒ "The Cat In The Hat"

(capitalize "THE 77TH-HATTED CAT")
     ⇒ "The 77th-Hatted Cat"

(capitalize ?x)
     ⇒ 88
```

`upcase-initials` *string-or-char* [Function]

If *string-or-char* is a string, this function capitalizes the initials of the words in *string-or-char*, without altering any letters other than the initials. It returns a new string whose contents are a copy of *string-or-char*, in which each word has had its initial letter converted to upper case.

The definition of a word is any sequence of consecutive characters that are assigned to the word constituent syntax class in the current syntax table (see Section 36.2.1 [Syntax Class Table], page 1026).

When the argument to `upcase-initials` is a character, `upcase-initials` has the same result as `upcase`.

```
(upcase-initials "The CAT in the hAt")
     ⇒ "The CAT In The HAt"
```

Note that case conversion is not a one-to-one mapping of codepoints and length of the result may differ from length of the argument. Furthermore, because passing a character forces return type to be a character, functions are unable to perform proper substitution and result may differ compared to treating a one-character string. For example:

```
(upcase "ﬁ")  ; note: single character, ligature "fi"
     ⇒ "FI"
(upcase ?ﬁ)
     ⇒ 64257  ; i.e. ?fi
```

To avoid this, a character must first be converted into a string, using `string` function, before being passed to one of the casing functions. Of course, no assumptions on the length of the result may be made.

Mapping for such special cases are taken from `special-uppercase`, `special-lowercase` and `special-titlecase` See Section 34.6 [Character Properties], page 971.

See Section 4.5 [Text Comparison], page 60, for functions that compare strings; some of them ignore case differences, or can optionally ignore case differences.

4.10 The Case Table

You can customize case conversion by installing a special *case table*. A case table specifies the mapping between upper case and lower case letters. It affects both the case conversion functions for Lisp objects (see the previous section) and those that apply to text in the buffer (see Section 33.18 [Case Changes], page 918). Each buffer has a case table; there is also a standard case table which is used to initialize the case table of new buffers.

A case table is a char-table (see Section 6.6 [Char-Tables], page 119) whose subtype is `case-table`. This char-table maps each character into the corresponding lower case character. It has three extra slots, which hold related tables:

upcase The upcase table maps each character into the corresponding upper case character.

canonicalize
The canonicalize table maps all of a set of case-related characters into a particular member of that set.

equivalences
The equivalences table maps each one of a set of case-related characters into the next character in that set.

In simple cases, all you need to specify is the mapping to lower-case; the three related tables will be calculated automatically from that one.

For some languages, upper and lower case letters are not in one-to-one correspondence. There may be two different lower case letters with the same upper case equivalent. In these cases, you need to specify the maps for both lower case and upper case.

The extra table *canonicalize* maps each character to a canonical equivalent; any two characters that are related by case-conversion have the same canonical equivalent character. For example, since '`a`' and '`A`' are related by case-conversion, they should have the same canonical equivalent character (which should be either '`a`' for both of them, or '`A`' for both of them).

The extra table *equivalences* is a map that cyclically permutes each equivalence class (of characters with the same canonical equivalent). (For ordinary ASCII, this would map '`a`' into '`A`' and '`A`' into '`a`', and likewise for each set of equivalent characters.)

When constructing a case table, you can provide `nil` for *canonicalize*; then Emacs fills in this slot from the lower case and upper case mappings. You can also provide `nil` for *equivalences*; then Emacs fills in this slot from *canonicalize*. In a case table that is actually in use, those components are non-`nil`. Do not try to specify *equivalences* without also specifying *canonicalize*.

Here are the functions for working with case tables:

`case-table-p` *object* [Function]
This predicate returns non-`nil` if *object* is a valid case table.

`set-standard-case-table` *table* [Function]
This function makes *table* the standard case table, so that it will be used in any buffers created subsequently.

`standard-case-table` [Function]
This returns the standard case table.

`current-case-table` [Function]
This function returns the current buffer's case table.

`set-case-table` *table* [Function]
This sets the current buffer's case table to *table*.

`with-case-table` *table body...* [Macro]
: The `with-case-table` macro saves the current case table, makes *table* the current case table, evaluates the *body* forms, and finally restores the case table. The return value is the value of the last form in *body*. The case table is restored even in case of an abnormal exit via `throw` or error (see Section 11.7 [Nonlocal Exits], page 177).

Some language environments modify the case conversions of ASCII characters; for example, in the Turkish language environment, the ASCII capital I is downcased into a Turkish dotless i ('`ı`'). This can interfere with code that requires ordinary ASCII case conversion, such as implementations of ASCII-based network protocols. In that case, use the `with-case-table` macro with the variable *ascii-case-table*, which stores the unmodified case table for the ASCII character set.

`ascii-case-table` [Variable]
: The case table for the ASCII character set. This should not be modified by any language environment settings.

The following three functions are convenient subroutines for packages that define non-ASCII character sets. They modify the specified case table *case-table*; they also modify the standard syntax table. See Chapter 36 [Syntax Tables], page 1025. Normally you would use these functions to change the standard case table.

`set-case-syntax-pair` *uc lc case-table* [Function]
: This function specifies a pair of corresponding letters, one upper case and one lower case.

`set-case-syntax-delims` *l r case-table* [Function]
: This function makes characters *l* and *r* a matching pair of case-invariant delimiters.

`set-case-syntax` *char syntax case-table* [Function]
: This function makes *char* case-invariant, with syntax *syntax*.

`describe-buffer-case-table` [Command]
: This command displays a description of the contents of the current buffer's case table.

5 Lists

A *list* represents a sequence of zero or more elements (which may be any Lisp objects). The important difference between lists and vectors is that two or more lists can share part of their structure; in addition, you can insert or delete elements in a list without copying the whole list.

5.1 Lists and Cons Cells

Lists in Lisp are not a primitive data type; they are built up from *cons cells* (see Section 2.4.6 [Cons Cell Type], page 16). A cons cell is a data object that represents an ordered pair. That is, it has two slots, and each slot *holds*, or *refers to*, some Lisp object. One slot is known as the CAR, and the other is known as the CDR. (These names are traditional; see Section 2.4.6 [Cons Cell Type], page 16.) CDR is pronounced "could-er".

We say that "the CAR of this cons cell is" whatever object its CAR slot currently holds, and likewise for the CDR.

A list is a series of cons cells chained together, so that each cell refers to the next one. There is one cons cell for each element of the list. By convention, the CARs of the cons cells hold the elements of the list, and the CDRs are used to chain the list (this asymmetry between CAR and CDR is entirely a matter of convention; at the level of cons cells, the CAR and CDR slots have similar properties). Hence, the CDR slot of each cons cell in a list refers to the following cons cell.

Also by convention, the CDR of the last cons cell in a list is `nil`. We call such a `nil`-terminated structure a *proper list*[1]. In Emacs Lisp, the symbol `nil` is both a symbol and a list with no elements. For convenience, the symbol `nil` is considered to have `nil` as its CDR (and also as its CAR).

Hence, the CDR of a proper list is always a proper list. The CDR of a nonempty proper list is a proper list containing all the elements except the first.

If the CDR of a list's last cons cell is some value other than `nil`, we call the structure a *dotted list*, since its printed representation would use dotted pair notation (see Section 2.4.6.2 [Dotted Pair Notation], page 18). There is one other possibility: some cons cell's CDR could point to one of the previous cons cells in the list. We call that structure a *circular list*.

For some purposes, it does not matter whether a list is proper, circular or dotted. If a program doesn't look far enough down the list to see the CDR of the final cons cell, it won't care. However, some functions that operate on lists demand proper lists and signal errors if given a dotted list. Most functions that try to find the end of a list enter infinite loops if given a circular list. You can use the function `proper-list-p`, described in the next section (see Section 5.2 [List-related Predicates], page 78), to determine whether a list is a proper one.

Because most cons cells are used as part of lists, we refer to any structure made out of cons cells as a *list structure*.

[1] It is sometimes also referred to as a *true list*, but we generally do not use this terminology in this manual.

5.2 Predicates on Lists

The following predicates test whether a Lisp object is an atom, whether it is a cons cell or is a list, or whether it is the distinguished object `nil`. (Many of these predicates can be defined in terms of the others, but they are used so often that it is worth having them.)

`consp` *object* [Function]
: This function returns `t` if *object* is a cons cell, `nil` otherwise. `nil` is not a cons cell, although it *is* a list.

`atom` *object* [Function]
: This function returns `t` if *object* is an atom, `nil` otherwise. All objects except cons cells are atoms. The symbol `nil` is an atom and is also a list; it is the only Lisp object that is both.

```
(atom object) ≡ (not (consp object))
```

`listp` *object* [Function]
: This function returns `t` if *object* is a cons cell or `nil`. Otherwise, it returns `nil`.

```
(listp '(1))
     ⇒ t
(listp '())
     ⇒ t
```

`nlistp` *object* [Function]
: This function is the opposite of `listp`: it returns `t` if *object* is not a list. Otherwise, it returns `nil`.

```
(listp object) ≡ (not (nlistp object))
```

`null` *object* [Function]
: This function returns `t` if *object* is `nil`, and returns `nil` otherwise. This function is identical to `not`, but as a matter of clarity we use `not` when *object* is considered a truth value (see `not` in Section 11.3 [Combining Conditions], page 162) and `null` otherwise.

```
(null '(1))
     ⇒ nil
(null '())
     ⇒ t
```

`proper-list-p` *object* [Function]
: This function returns the length of *object* if it is a proper list, `nil` otherwise (see Section 5.1 [Cons Cells], page 77). In addition to satisfying `listp`, a proper list is neither circular nor dotted.

```
(proper-list-p '(a b c))
    ⇒ 3
(proper-list-p '(a b . c))
    ⇒ nil
```

5.3 Accessing Elements of Lists

`car` *cons-cell* [Function]

This function returns the value referred to by the first slot of the cons cell *cons-cell*. In other words, it returns the CAR of *cons-cell*.

As a special case, if *cons-cell* is `nil`, this function returns `nil`. Therefore, any list is a valid argument. An error is signaled if the argument is not a cons cell or `nil`.

```
(car '(a b c))
     ⇒ a
(car '())
     ⇒ nil
```

`cdr` *cons-cell* [Function]

This function returns the value referred to by the second slot of the cons cell *cons-cell*. In other words, it returns the CDR of *cons-cell*.

As a special case, if *cons-cell* is `nil`, this function returns `nil`; therefore, any list is a valid argument. An error is signaled if the argument is not a cons cell or `nil`.

```
(cdr '(a b c))
     ⇒ (b c)
(cdr '())
     ⇒ nil
```

`car-safe` *object* [Function]

This function lets you take the CAR of a cons cell while avoiding errors for other data types. It returns the CAR of *object* if *object* is a cons cell, `nil` otherwise. This is in contrast to `car`, which signals an error if *object* is not a list.

```
(car-safe object)
≡
(let ((x object))
  (if (consp x)
      (car x)
    nil))
```

`cdr-safe` *object* [Function]

This function lets you take the CDR of a cons cell while avoiding errors for other data types. It returns the CDR of *object* if *object* is a cons cell, `nil` otherwise. This is in contrast to `cdr`, which signals an error if *object* is not a list.

```
(cdr-safe object)
≡
(let ((x object))
  (if (consp x)
      (cdr x)
    nil))
```

`pop` *listname* [Macro]

This macro provides a convenient way to examine the CAR of a list, and take it off the list, all at once. It operates on the list stored in *listname*. It removes the first element from the list, saves the CDR into *listname*, then returns the removed element.

In the simplest case, *listname* is an unquoted symbol naming a list; in that case, this macro is equivalent to `(prog1 (car listname) (setq listname (cdr listname)))`.

```
x
     ⇒ (a b c)
(pop x)
     ⇒ a
x
     ⇒ (b c)
```

More generally, *listname* can be a generalized variable. In that case, this macro saves into *listname* using `setf`. See Section 12.17 [Generalized Variables], page 224.

For the `push` macro, which adds an element to a list, See Section 5.5 [List Variables], page 85.

`nth` *n list* [Function]

This function returns the *n*th element of *list*. Elements are numbered starting with zero, so the CAR of *list* is element number zero. If the length of *list* is *n* or less, the value is `nil`.

```
(nth 2 '(1 2 3 4))
     ⇒ 3
(nth 10 '(1 2 3 4))
     ⇒ nil

(nth n x) ≡ (car (nthcdr n x))
```

The function `elt` is similar, but applies to any kind of sequence. For historical reasons, it takes its arguments in the opposite order. See Section 6.1 [Sequence Functions], page 102.

`nthcdr` *n list* [Function]

This function returns the *n*th CDR of *list*. In other words, it skips past the first *n* links of *list* and returns what follows.

If *n* is zero, `nthcdr` returns all of *list*. If the length of *list* is *n* or less, `nthcdr` returns `nil`.

```
(nthcdr 1 '(1 2 3 4))
     ⇒ (2 3 4)
(nthcdr 10 '(1 2 3 4))
     ⇒ nil
(nthcdr 0 '(1 2 3 4))
     ⇒ (1 2 3 4)
```

`take` *n list* [Function]

This function returns the *n* first elements of *list*. Essentially, it returns the part of *list* that `nthcdr` skips.

`take` returns *list* if shorter than *n* elements; it returns `nil` if *n* is zero or negative.

```
(take 3 '(a b c d))
     ⇒ (a b c)
```

```
(take 10 '(a b c d))
     ⇒ (a b c d)
(take 0 '(a b c d))
     ⇒ nil
```

`ntake` *n list* [Function]

This is a version of `take` that works by destructively modifying the list structure of the argument. That makes it faster, but the original value of *list* may be lost.

`ntake` returns *list* unmodified if shorter than *n* elements; it returns `nil` if *n* is zero or negative. Otherwise, it returns *list* truncated to its first *n* elements.

This means that it is usually a good idea to use the return value and not just rely on the truncation effect unless *n* is known to be positive.

`last` *list* **&optional** *n* [Function]

This function returns the last link of *list*. The `car` of this link is the list's last element. If *list* is null, `nil` is returned. If *n* is non-`nil`, the *n*th-to-last link is returned instead, or the whole of *list* if *n* is bigger than *list*'s length.

`safe-length` *list* [Function]

This function returns the length of *list*, with no risk of either an error or an infinite loop. It generally returns the number of distinct cons cells in the list. However, for circular lists, the value is just an upper bound; it is often too large.

If *list* is not `nil` or a cons cell, `safe-length` returns 0.

The most common way to compute the length of a list, when you are not worried that it may be circular, is with `length`. See Section 6.1 [Sequence Functions], page 102.

`caar` *cons-cell* [Function]

This is the same as `(car (car` *cons-cell*`))`.

`cadr` *cons-cell* [Function]

This is the same as `(car (cdr` *cons-cell*`))` or `(nth 1` *cons-cell*`)`.

`cdar` *cons-cell* [Function]

This is the same as `(cdr (car` *cons-cell*`))`.

`cddr` *cons-cell* [Function]

This is the same as `(cdr (cdr` *cons-cell*`))` or `(nthcdr 2` *cons-cell*`)`.

In addition to the above, 24 additional compositions of `car` and `cdr` are defined as `cxxxr` and `cxxxxr`, where each *x* is either `a` or `d`. `cadr`, `caddr`, and `cadddr` pick out the second, third or fourth elements of a list, respectively. `cl-lib` provides the same under the names `cl-second`, `cl-third`, and `cl-fourth`. See Section "List Functions" in *Common Lisp Extensions*.

`butlast` *x* **&optional** *n* [Function]

This function returns the list *x* with the last element, or the last *n* elements, removed. If *n* is greater than zero it makes a copy of the list so as not to damage the original list. In general, `(append (butlast` *x n*`) (last` *x n*`))` will return a list equal to *x*.

`nbutlast` *x* **&optional** *n* [Function]
This is a version of `butlast` that works by destructively modifying the `cdr` of the appropriate element, rather than making a copy of the list.

5.4 Building Cons Cells and Lists

Many functions build lists, as lists reside at the very heart of Lisp. `cons` is the fundamental list-building function; however, it is interesting to note that `list` is used more times in the source code for Emacs than `cons`.

`cons` *object1 object2* [Function]
This function is the most basic function for building new list structure. It creates a new cons cell, making *object1* the CAR, and *object2* the CDR. It then returns the new cons cell. The arguments *object1* and *object2* may be any Lisp objects, but most often *object2* is a list.

```
(cons 1 '(2))
     ⇒ (1 2)
(cons 1 '())
     ⇒ (1)
(cons 1 2)
     ⇒ (1 . 2)
```

`cons` is often used to add a single element to the front of a list. This is called *consing the element onto the list*.[2] For example:

```
(setq list (cons newelt list))
```

Note that there is no conflict between the variable named `list` used in this example and the function named `list` described below; any symbol can serve both purposes.

`list` **&rest** *objects* [Function]
This function creates a list with *objects* as its elements. The resulting list is always `nil`-terminated. If no *objects* are given, the empty list is returned.

```
(list 1 2 3 4 5)
     ⇒ (1 2 3 4 5)
(list 1 2 '(3 4 5) 'foo)
     ⇒ (1 2 (3 4 5) foo)
(list)
     ⇒ nil
```

`make-list` *length object* [Function]
This function creates a list of *length* elements, in which each element is *object*. Compare `make-list` with `make-string` (see Section 4.3 [Creating Strings], page 55).

```
(make-list 3 'pigs)
     ⇒ (pigs pigs pigs)
```

[2] There is no strictly equivalent way to add an element to the end of a list. You can use `(append` *listname* `(list` *newelt*`))`, which creates a whole new list by copying *listname* and adding *newelt* to its end. Or you can use `(nconc` *listname* `(list` *newelt*`))`, which modifies *listname* by following all the CDRs and then replacing the terminating `nil`. Compare this to adding an element to the beginning of a list with `cons`, which neither copies nor modifies the list.

```
(make-list 0 'pigs)
     ⇒ nil
(setq l (make-list 3 '(a b)))
     ⇒ ((a b) (a b) (a b))
(eq (car l) (cadr l))
     ⇒ t
```

`append` **&rest** *sequences* [Function]

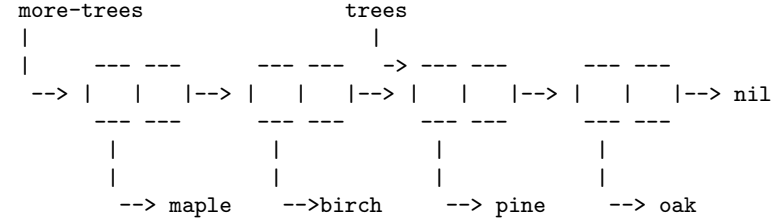
This function returns a list containing all the elements of *sequences*. The *sequences* may be lists, vectors, bool-vectors, or strings, but the last one should usually be a list. All arguments except the last one are copied, so none of the arguments is altered. (See `nconc` in Section 5.6.3 [Rearrangement], page 90, for a way to join lists with no copying.)

More generally, the final argument to `append` may be any Lisp object. The final argument is not copied or converted; it becomes the CDR of the last cons cell in the new list. If the final argument is itself a list, then its elements become in effect elements of the result list. If the final element is not a list, the result is a dotted list since its final CDR is not `nil` as required in a proper list (see Section 5.1 [Cons Cells], page 77).

Here is an example of using `append`:

```
(setq trees '(pine oak))
     ⇒ (pine oak)
(setq more-trees (append '(maple birch) trees))
     ⇒ (maple birch pine oak)

trees
     ⇒ (pine oak)
more-trees
     ⇒ (maple birch pine oak)
(eq trees (cdr (cdr more-trees)))
     ⇒ t
```

You can see how `append` works by looking at a box diagram. The variable `trees` is set to the list `(pine oak)` and then the variable `more-trees` is set to the list `(maple birch pine oak)`. However, the variable `trees` continues to refer to the original list:



An empty sequence contributes nothing to the value returned by `append`. As a consequence of this, a final `nil` argument forces a copy of the previous argument:

```
trees
     ⇒ (pine oak)
```

```
(setq wood (append trees nil))
     ⇒ (pine oak)
wood
     ⇒ (pine oak)
(eq wood trees)
     ⇒ nil
```

This once was the usual way to copy a list, before the function `copy-sequence` was invented. See Chapter 6 [Sequences Arrays Vectors], page 102.

Here we show the use of vectors and strings as arguments to `append`:

```
(append [a b] "cd" nil)
     ⇒ (a b 99 100)
```

With the help of `apply` (see Section 13.5 [Calling Functions], page 239), we can append all the lists in a list of lists:

```
(apply 'append '((a b c) nil (x y z) nil))
     ⇒ (a b c x y z)
```

If no *sequences* are given, `nil` is returned:

```
(append)
     ⇒ nil
```

Here are some examples where the final argument is not a list:

```
(append '(x y) 'z)
     ⇒ (x y . z)
(append '(x y) [z])
     ⇒ (x y . [z])
```

The second example shows that when the final argument is a sequence but not a list, the sequence's elements do not become elements of the resulting list. Instead, the sequence becomes the final CDR, like any other non-list final argument.

`copy-tree` *tree* **&optional** *vecp* [Function]

This function returns a copy of the tree *tree*. If *tree* is a cons cell, this makes a new cons cell with the same CAR and CDR, then recursively copies the CAR and CDR in the same way.

Normally, when *tree* is anything other than a cons cell, `copy-tree` simply returns *tree*. However, if *vecp* is non-`nil`, it copies vectors too (and operates recursively on their elements).

`flatten-tree` *tree* [Function]

This function returns a "flattened" copy of *tree*, that is, a list containing all the non-`nil` terminal nodes, or leaves, of the tree of cons cells rooted at *tree*. Leaves in the returned list are in the same order as in *tree*.

```
(flatten-tree '(1 (2 . 3) nil (4 5 (6)) 7))
    ⇒(1 2 3 4 5 6 7)
```

`ensure-list` *object* [Function]

This function returns *object* as a list. If *object* is already a list, the function returns it; otherwise, the function returns a one-element list containing *object*.

This is usually useful if you have a variable that may or may not be a list, and you can then say, for instance:

```
(dolist (elem (ensure-list foo))
  (princ elem))
```

`number-sequence` *from* **&optional** *to separation* [Function]

This function returns a list of numbers starting with *from* and incrementing by *separation*, and ending at or just before *to*. *separation* can be positive or negative and defaults to 1. If *to* is `nil` or numerically equal to *from*, the value is the one-element list `(`*from*`)`. If *to* is less than *from* with a positive *separation*, or greater than *from* with a negative *separation*, the value is `nil` because those arguments specify an empty sequence.

If *separation* is 0 and *to* is neither `nil` nor numerically equal to *from*, `number-sequence` signals an error, since those arguments specify an infinite sequence.

All arguments are numbers. Floating-point arguments can be tricky, because floating-point arithmetic is inexact. For instance, depending on the machine, it may quite well happen that `(number-sequence 0.4 0.6 0.2)` returns the one element list `(0.4)`, whereas `(number-sequence 0.4 0.8 0.2)` returns a list with three elements. The *n*th element of the list is computed by the exact formula `(+ `*from*` (* `*n*` `*separation*`))`. Thus, if one wants to make sure that *to* is included in the list, one can pass an expression of this exact type for *to*. Alternatively, one can replace *to* with a slightly larger value (or a slightly more negative value if *separation* is negative).

Some examples:

```
(number-sequence 4 9)
     ⇒ (4 5 6 7 8 9)
(number-sequence 9 4 -1)
     ⇒ (9 8 7 6 5 4)
(number-sequence 9 4 -2)
     ⇒ (9 7 5)
(number-sequence 8)
     ⇒ (8)
(number-sequence 8 5)
     ⇒ nil
(number-sequence 5 8 -1)
     ⇒ nil
(number-sequence 1.5 6 2)
     ⇒ (1.5 3.5 5.5)
```

5.5 Modifying List Variables

These functions, and one macro, provide convenient ways to modify a list which is stored in a variable.

`push` *element listname* [Macro]

This macro creates a new list whose CAR is *element* and whose CDR is the list specified by *listname*, and saves that list in *listname*. In the simplest case,

listname is an unquoted symbol naming a list, and this macro is equivalent to `(setq listname (cons element listname))`.

```
(setq l '(a b))
     ⇒ (a b)
(push 'c l)
     ⇒ (c a b)
l
     ⇒ (c a b)
```

More generally, `listname` can be a generalized variable. In that case, this macro does the equivalent of `(setf listname (cons element listname))`. See Section 12.17 [Generalized Variables], page 224.

For the `pop` macro, which removes the first element from a list, See Section 5.3 [List Elements], page 79.

Two functions modify lists that are the values of variables.

`add-to-list` *symbol element* **&optional** *append compare-fn* [Function]

This function sets the variable *symbol* by consing *element* onto the old value, if *element* is not already a member of that value. It returns the resulting list, whether updated or not. The value of *symbol* had better be a list already before the call. `add-to-list` uses *compare-fn* to compare *element* against existing list members; if *compare-fn* is `nil`, it uses `equal`.

Normally, if *element* is added, it is added to the front of *symbol*, but if the optional argument *append* is non-`nil`, it is added at the end.

The argument *symbol* is not implicitly quoted; `add-to-list` is an ordinary function, like `set` and unlike `setq`. Quote the argument yourself if that is what you want.

Do not use this function when *symbol* refers to a lexical variable.

Here's a scenario showing how to use `add-to-list`:

```
(setq foo '(a b))
     ⇒ (a b)

(add-to-list 'foo 'c)     ;; Add c.
     ⇒ (c a b)

(add-to-list 'foo 'b)     ;; No effect.
     ⇒ (c a b)

foo                       ;; foo was changed.
     ⇒ (c a b)
```

An equivalent expression for `(add-to-list 'var value)` is this:

```
(if (member value var)
    var
  (setq var (cons value var)))
```

`add-to-ordered-list` *symbol element* **&optional** *order* [Function]

This function sets the variable *symbol* by inserting *element* into the old value, which must be a list, at the position specified by *order*. If *element* is already a member of the list, its position in the list is adjusted according to *order*. Membership is tested using `eq`. This function returns the resulting list, whether updated or not.

The *order* is typically a number (integer or float), and the elements of the list are sorted in non-decreasing numerical order.

order may also be omitted or `nil`. Then the numeric order of *element* stays unchanged if it already has one; otherwise, *element* has no numeric order. Elements without a numeric list order are placed at the end of the list, in no particular order.

Any other value for *order* removes the numeric order of *element* if it already has one; otherwise, it is equivalent to `nil`.

The argument *symbol* is not implicitly quoted; `add-to-ordered-list` is an ordinary function, like `set` and unlike `setq`. Quote the argument yourself if necessary.

The ordering information is stored in a hash table on *symbol*'s `list-order` property. *symbol* cannot refer to a lexical variable.

Here's a scenario showing how to use `add-to-ordered-list`:

```
(setq foo '())
     ⇒ nil

(add-to-ordered-list 'foo 'a 1)     ;; Add a.
     ⇒ (a)

(add-to-ordered-list 'foo 'c 3)     ;; Add c.
     ⇒ (a c)

(add-to-ordered-list 'foo 'b 2)     ;; Add b.
     ⇒ (a b c)

(add-to-ordered-list 'foo 'b 4)     ;; Move b.
     ⇒ (a c b)

(add-to-ordered-list 'foo 'd)       ;; Append d.
     ⇒ (a c b d)

(add-to-ordered-list 'foo 'e)       ;; Add e.
     ⇒ (a c b e d)

foo                         ;; foo was changed.
     ⇒ (a c b e d)
```

5.6 Modifying Existing List Structure

You can modify the CAR and CDR contents of a cons cell with the primitives `setcar` and `setcdr`. These are destructive operations because they change existing list structure. Destructive operations should be applied only to mutable lists, that is, lists constructed via

`cons`, `list` or similar operations. Lists created by quoting are part of the program and should not be changed by destructive operations. See Section 2.9 [Mutability], page 37.

> **Common Lisp note:** Common Lisp uses functions `rplaca` and `rplacd` to alter list structure; they change structure the same way as `setcar` and `setcdr`, but the Common Lisp functions return the cons cell while `setcar` and `setcdr` return the new CAR or CDR.

5.6.1 Altering List Elements with `setcar`

Changing the CAR of a cons cell is done with `setcar`. When used on a list, `setcar` replaces one element of a list with a different element.

`setcar` *cons object* [Function]

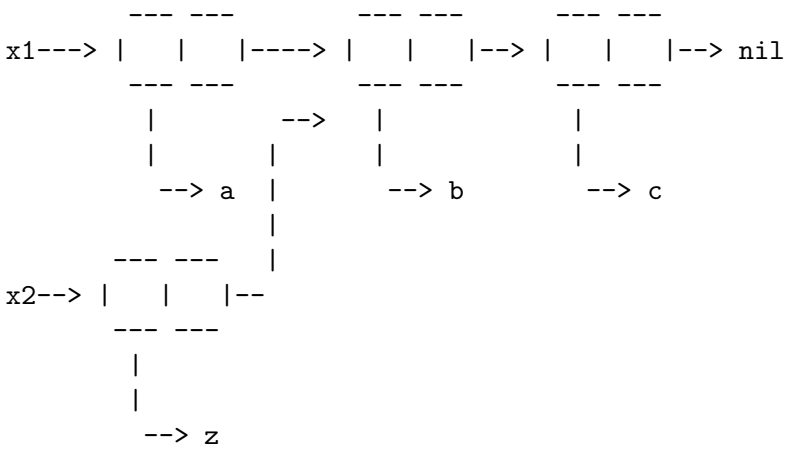
This function stores *object* as the new CAR of *cons*, replacing its previous CAR. In other words, it changes the CAR slot of *cons* to refer to *object*. It returns the value *object*. For example:

```
(setq x (list 1 2))
     ⇒ (1 2)
(setcar x 4)
     ⇒ 4
x
     ⇒ (4 2)
```

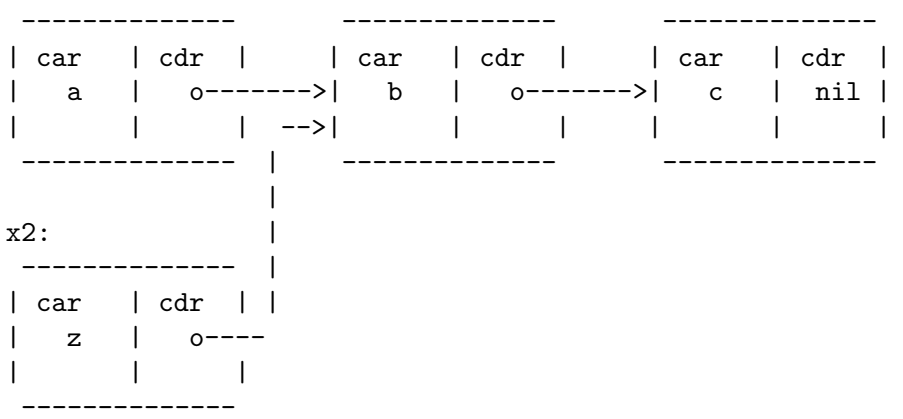
When a cons cell is part of the shared structure of several lists, storing a new CAR into the cons changes one element of each of these lists. Here is an example:

```
;; Create two lists that are partly shared.
(setq x1 (list 'a 'b 'c))
     ⇒ (a b c)
(setq x2 (cons 'z (cdr x1)))
     ⇒ (z b c)

;; Replace the CAR of a shared link.
(setcar (cdr x1) 'foo)
     ⇒ foo
x1                          ; Both lists are changed.
     ⇒ (a foo c)
x2
     ⇒ (z foo c)

;; Replace the CAR of a link that is not shared.
(setcar x1 'baz)
     ⇒ baz
x1                          ; Only one list is changed.
     ⇒ (baz foo c)
x2
     ⇒ (z foo c)
```

Here is a graphical depiction of the shared structure of the two lists in the variables `x1` and `x2`, showing why replacing `b` changes them both:



Here is an alternative form of box diagram, showing the same relationship:



5.6.2 Altering the CDR of a List

The lowest-level primitive for modifying a CDR is `setcdr`:

`setcdr` *cons object* [Function]

This function stores *object* as the new CDR of *cons*, replacing its previous CDR. In other words, it changes the CDR slot of *cons* to refer to *object*. It returns the value *object*.

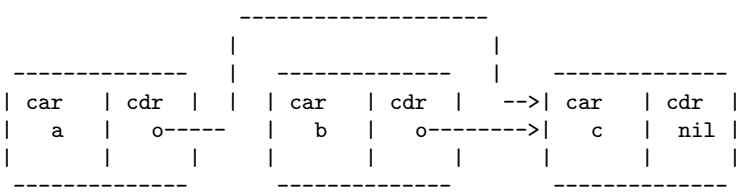
Here is an example of replacing the CDR of a list with a different list. All but the first element of the list are removed in favor of a different sequence of elements. The first element is unchanged, because it resides in the CAR of the list, and is not reached via the CDR.

```
(setq x (list 1 2 3))
     ⇒ (1 2 3)
(setcdr x '(4))
     ⇒ (4)
x
     ⇒ (1 4)
```

You can delete elements from the middle of a list by altering the CDRs of the cons cells in the list. For example, here we delete the second element, `b`, from the list `(a b c)`, by changing the CDR of the first cons cell:

```
(setq x1 (list 'a 'b 'c))
     ⇒ (a b c)
(setcdr x1 (cdr (cdr x1)))
     ⇒ (c)
x1
     ⇒ (a c)
```

Here is the result in box notation:

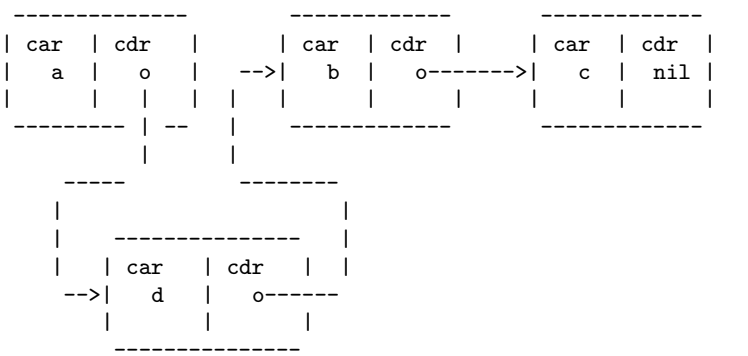


The second cons cell, which previously held the element `b`, still exists and its CAR is still `b`, but it no longer forms part of this list.

It is equally easy to insert a new element by changing CDRs:

```
(setq x1 (list 'a 'b 'c))
     ⇒ (a b c)
(setcdr x1 (cons 'd (cdr x1)))
     ⇒ (d b c)
x1
     ⇒ (a d b c)
```

Here is this result in box notation:



5.6.3 Functions that Rearrange Lists

Here are some functions that rearrange lists destructively by modifying the CDRs of their component cons cells. These functions are destructive because they chew up the original lists passed to them as arguments, relinking their cons cells to form a new list that is the returned value.

The function `delq` in the following section is another example of destructive list manipulation.

`nconc` **&rest** *lists* [Function]

This function returns a list containing all the elements of *lists*. Unlike `append` (see Section 5.4 [Building Lists], page 82), the *lists* are *not* copied. Instead, the last CDR of each of the *lists* is changed to refer to the following list. The last of the *lists* is not altered. For example:

```
(setq x (list 1 2 3))
     ⇒ (1 2 3)
(nconc x '(4 5))
     ⇒ (1 2 3 4 5)
x
     ⇒ (1 2 3 4 5)
```

Since the last argument of `nconc` is not itself modified, it is reasonable to use a constant list, such as `'(4 5)`, as in the above example. For the same reason, the last argument need not be a list:

```
(setq x (list 1 2 3))
     ⇒ (1 2 3)
(nconc x 'z)
     ⇒ (1 2 3 . z)
x
     ⇒ (1 2 3 . z)
```

However, the other arguments (all but the last) should be mutable lists.

A common pitfall is to use a constant list as a non-last argument to `nconc`. If you do this, the resulting behavior is undefined (see Section 10.1.1 [Self-Evaluating Forms], page 146). It is possible that your program will change each time you run it! Here is what might happen (though this is not guaranteed to happen):

```
(defun add-foo (x)             ; We want this function to add
  (nconc '(foo) x))            ;   foo to the front of its arg.

(symbol-function 'add-foo)
     ⇒ (lambda (x) (nconc '(foo) x))

(setq xx (add-foo '(1 2)))     ; It seems to work.
     ⇒ (foo 1 2)
(setq xy (add-foo '(3 4)))     ; What happened?
     ⇒ (foo 1 2 3 4)
(eq xx xy)
     ⇒ t

(symbol-function 'add-foo)
     ⇒ (lambda (x) (nconc '(foo 1 2 3 4) x))
```

5.7 Using Lists as Sets

A list can represent an unordered mathematical set—simply consider a value an element of a set if it appears in the list, and ignore the order of the list. To form the union of two sets, use `append` (as long as you don't mind having duplicate elements). You can remove `equal` duplicates using `delete-dups` or `seq-uniq`. Other useful functions for sets include `memq` and `delq`, and their `equal` versions, `member` and `delete`.

Common Lisp note: Common Lisp has functions `union` (which avoids duplicate elements) and `intersection` for set operations. In Emacs Lisp, variants of these facilities are provided by the `cl-lib` library. See Section "Lists as Sets" in *Common Lisp Extensions*.

`memq` *object list* [Function]

This function tests to see whether *object* is a member of *list*. If it is, `memq` returns a list starting with the first occurrence of *object*. Otherwise, it returns `nil`. The letter '`q`' in `memq` says that it uses `eq` to compare *object* against the elements of the list. For example:

```
(memq 'b '(a b c b a))
     ⇒ (b c b a)
(memq '(2) '((1) (2)))    ; The two (2)s need not be eq.
     ⇒ Unspecified; might be nil or ((2)).
```

`delq` *object list* [Function]

This function destructively removes all elements `eq` to *object* from *list*, and returns the resulting list. The letter '`q`' in `delq` says that it uses `eq` to compare *object* against the elements of the list, like `memq` and `remq`.

Typically, when you invoke `delq`, you should use the return value by assigning it to the variable which held the original list. The reason for this is explained below.

The `delq` function deletes elements from the front of the list by simply advancing down the list, and returning a sublist that starts after those elements. For example:

```
(delq 'a '(a b c)) ≡ (cdr '(a b c))
```

When an element to be deleted appears in the middle of the list, removing it involves changing the CDRs (see Section 5.6.2 [Setcdr], page 89).

```
(setq sample-list (list 'a 'b 'c '(4)))
     ⇒ (a b c (4))
(delq 'a sample-list)
     ⇒ (b c (4))
sample-list
     ⇒ (a b c (4))
(delq 'c sample-list)
     ⇒ (a b (4))
sample-list
     ⇒ (a b (4))
```

Note that `(delq 'c sample-list)` modifies `sample-list` to splice out the third element, but `(delq 'a sample-list)` does not splice anything—it just returns a shorter list. Don't assume that a variable which formerly held the argument *list* now has fewer elements, or that it still holds the original list! Instead, save the result of `delq` and use that. Most often we store the result back into the variable that held the original list:

```
(setq flowers (delq 'rose flowers))
```

In the following example, the `(list 4)` that `delq` attempts to match and the `(4)` in the `sample-list` are `equal` but not `eq`:

```
(delq (list 4) sample-list)
     ⇒ (a c (4))
```

If you want to delete elements that are `equal` to a given value, use `delete` (see below).

`remq` *object list* [Function]

This function returns a copy of *list*, with all elements removed which are `eq` to *object*. The letter 'q' in `remq` says that it uses `eq` to compare *object* against the elements of `list`.

```
(setq sample-list (list 'a 'b 'c 'a 'b 'c))
     ⇒ (a b c a b c)
(remq 'a sample-list)
     ⇒ (b c b c)
sample-list
     ⇒ (a b c a b c)
```

`memql` *object list* [Function]

The function `memql` tests to see whether *object* is a member of *list*, comparing members with *object* using `eql`, so floating-point elements are compared by value. If *object* is a member, `memql` returns a list starting with its first occurrence in *list*. Otherwise, it returns `nil`.

Compare this with `memq`:

```
(memql 1.2 '(1.1 1.2 1.3))  ; 1.2 and 1.2 are eql.
     ⇒ (1.2 1.3)
(memq 1.2 '(1.1 1.2 1.3))  ; The two 1.2s need not be eq.
     ⇒ Unspecified; might be nil or (1.2 1.3).
```

The following three functions are like `memq`, `delq` and `remq`, but use `equal` rather than `eq` to compare elements. See Section 2.8 [Equality Predicates], page 34.

`member` *object list* [Function]

The function `member` tests to see whether *object* is a member of *list*, comparing members with *object* using `equal`. If *object* is a member, `member` returns a list starting with its first occurrence in *list*. Otherwise, it returns `nil`.

Compare this with `memq`:

```
(member '(2) '((1) (2)))  ; (2) and (2) are equal.
     ⇒ ((2))
(memq '(2) '((1) (2)))    ; The two (2)s need not be eq.
     ⇒ Unspecified; might be nil or (2).
;; Two strings with the same contents are equal.
(member "foo" '("foo" "bar"))
     ⇒ ("foo" "bar")
```

`delete` *object sequence* [Function]

This function removes all elements `equal` to *object* from *sequence*, and returns the resulting sequence.

If *sequence* is a list, `delete` is to `delq` as `member` is to `memq`: it uses `equal` to compare elements with *object*, like `member`; when it finds an element that matches, it cuts the element out just as `delq` would. As with `delq`, you should typically use the return value by assigning it to the variable which held the original list.

If `sequence` is a vector or string, `delete` returns a copy of `sequence` with all elements `equal` to `object` removed.

For example:

```
(setq l (list '(2) '(1) '(2)))
(delete '(2) l)
     ⇒ ((1))
l
     ⇒ ((2) (1))
;; If you want to change l reliably,
;; write (setq l (delete '(2) l)).
(setq l (list '(2) '(1) '(2)))
(delete '(1) l)
     ⇒ ((2) (2))
l
     ⇒ ((2) (2))
;; In this case, it makes no difference whether you set l,
;; but you should do so for the sake of the other case.
(delete '(2) [(2) (1) (2)])
     ⇒ [(1)]
```

`remove` *object sequence* [Function]

This function is the non-destructive counterpart of `delete`. It returns a copy of `sequence`, a list, vector, or string, with elements `equal` to `object` removed. For example:

```
(remove '(2) '((2) (1) (2)))
     ⇒ ((1))
(remove '(2) [(2) (1) (2)])
     ⇒ [(1)]
```

Common Lisp note: The functions `member`, `delete` and `remove` in GNU Emacs Lisp are derived from Maclisp, not Common Lisp. The Common Lisp versions do not use `equal` to compare elements.

`member-ignore-case` *object list* [Function]

This function is like `member`, except that *object* should be a string and that it ignores differences in letter-case and text representation: upper-case and lower-case letters are treated as equal, and unibyte strings are converted to multibyte prior to comparison.

`delete-dups` *list* [Function]

This function destructively removes all `equal` duplicates from *list*, stores the result in *list* and returns it. Of several `equal` occurrences of an element in *list*, `delete-dups` keeps the first one. See `seq-uniq` for non-destructive operation (see Section 6.1 [Sequence Functions], page 102).

See also the function `add-to-list`, in Section 5.5 [List Variables], page 85, for a way to add an element to a list stored in a variable and used as a set.

5.8 Association Lists

An *association list*, or *alist* for short, records a mapping from keys to values. It is a list of cons cells called *associations*: the CAR of each cons cell is the *key*, and the CDR is the *associated value*.[3]

Here is an example of an alist. The key `pine` is associated with the value `cones`; the key `oak` is associated with `acorns`; and the key `maple` is associated with `seeds`.

```
((pine . cones)
 (oak . acorns)
 (maple . seeds))
```

Both the values and the keys in an alist may be any Lisp objects. For example, in the following alist, the symbol `a` is associated with the number `1`, and the string `"b"` is associated with the *list* `(2 3)`, which is the CDR of the alist element:

```
((a . 1) ("b" 2 3))
```

Sometimes it is better to design an alist to store the associated value in the CAR of the CDR of the element. Here is an example of such an alist:

```
((rose red) (lily white) (buttercup yellow))
```

Here we regard `red` as the value associated with `rose`. One advantage of this kind of alist is that you can store other related information—even a list of other items—in the CDR of the CDR. One disadvantage is that you cannot use `rassq` (see below) to find the element containing a given value. When neither of these considerations is important, the choice is a matter of taste, as long as you are consistent about it for any given alist.

The same alist shown above could be regarded as having the associated value in the CDR of the element; the value associated with `rose` would be the list `(red)`.

Association lists are often used to record information that you might otherwise keep on a stack, since new associations may be added easily to the front of the list. When searching an association list for an association with a given key, the first one found is returned, if there is more than one.

In Emacs Lisp, it is *not* an error if an element of an association list is not a cons cell. The alist search functions simply ignore such elements. Many other versions of Lisp signal errors in such cases.

Note that property lists are similar to association lists in several respects. A property list behaves like an association list in which each key can occur only once. See Section 5.9 [Property Lists], page 99, for a comparison of property lists and association lists.

`assoc` *key alist* **&optional** *testfn* [Function]

This function returns the first association for *key* in *alist*, comparing *key* against the alist elements using *testfn* if it is a function, and `equal` otherwise (see Section 2.8 [Equality Predicates], page 34). If *testfn* is a function, it is called with two arguments: the CAR of an element from *alist* and *key*. The function returns `nil` if no association in *alist* has a CAR equal to *key*, as tested by *testfn*. For example:

```
(setq trees '((pine . cones) (oak . acorns) (maple . seeds)))
     ⇒ ((pine . cones) (oak . acorns) (maple . seeds))
```

[3] This usage of "key" is not related to the term "key sequence"; it means a value used to look up an item in a table. In this case, the table is the alist, and the alist associations are the items.

```
(assoc 'oak trees)
     ⇒ (oak . acorns)
(cdr (assoc 'oak trees))
     ⇒ acorns
(assoc 'birch trees)
     ⇒ nil
```

Here is another example, in which the keys and values are not symbols:

```
(setq needles-per-cluster
      '((2 "Austrian Pine" "Red Pine")
        (3 "Pitch Pine")
        (5 "White Pine")))

(cdr (assoc 3 needles-per-cluster))
     ⇒ ("Pitch Pine")
(cdr (assoc 2 needles-per-cluster))
     ⇒ ("Austrian Pine" "Red Pine")
```

The function `assoc-string` is much like `assoc` except that it ignores certain differences between strings. See Section 4.5 [Text Comparison], page 60.

`rassoc` *value alist* [Function]

This function returns the first association with value *value* in *alist*. It returns `nil` if no association in *alist* has a CDR `equal` to *value*.

`rassoc` is like `assoc` except that it compares the CDR of each *alist* association instead of the CAR. You can think of this as reverse `assoc`, finding the key for a given value.

`assq` *key alist* [Function]

This function is like `assoc` in that it returns the first association for *key* in *alist*, but it makes the comparison using `eq`. `assq` returns `nil` if no association in *alist* has a CAR `eq` to *key*. This function is used more often than `assoc`, since `eq` is faster than `equal` and most alists use symbols as keys. See Section 2.8 [Equality Predicates], page 34.

```
(setq trees '((pine . cones) (oak . acorns) (maple . seeds)))
     ⇒ ((pine . cones) (oak . acorns) (maple . seeds))
(assq 'pine trees)
     ⇒ (pine . cones)
```

On the other hand, `assq` is not usually useful in alists where the keys may not be symbols:

```
(setq leaves
      '(("simple leaves" . oak)
        ("compound leaves" . horsechestnut)))

(assq "simple leaves" leaves)
     ⇒ Unspecified; might be nil or ("simple leaves" . oak).
(assoc "simple leaves" leaves)
     ⇒ ("simple leaves" . oak)
```

`alist-get` *key alist* **&optional** *default remove testfn* [Function]

This function is similar to `assq`. It finds the first association `(`*key* `.` *value*`)` by comparing *key* with *alist* elements, and, if found, returns the *value* of that association. If no association is found, the function returns *default*. Comparison of *key* against *alist* elements uses the function specified by *testfn*, defaulting to `eq`.

This is a generalized variable (see Section 12.17 [Generalized Variables], page 224) that can be used to change a value with `setf`. When using it to set a value, optional argument *remove* non-`nil` means to remove *key*'s association from *alist* if the new value is `eql` to *default*.

`rassq` *value alist* [Function]

This function returns the first association with value *value* in *alist*. It returns `nil` if no association in *alist* has a CDR `eq` to *value*.

`rassq` is like `assq` except that it compares the CDR of each *alist* association instead of the CAR. You can think of this as reverse `assq`, finding the key for a given value.

For example:

```
(setq trees '((pine . cones) (oak . acorns) (maple . seeds)))

(rassq 'acorns trees)
     ⇒ (oak . acorns)
(rassq 'spores trees)
     ⇒ nil
```

`rassq` cannot search for a value stored in the CAR of the CDR of an element:

```
(setq colors '((rose red) (lily white) (buttercup yellow)))

(rassq 'white colors)
     ⇒ nil
```

In this case, the CDR of the association `(lily white)` is not the symbol `white`, but rather the list `(white)`. This becomes clearer if the association is written in dotted pair notation:

```
(lily white) ≡ (lily . (white))
```

`assoc-default` *key alist* **&optional** *test default* [Function]

This function searches *alist* for a match for *key*. For each element of *alist*, it compares the element (if it is an atom) or the element's CAR (if it is a cons) against *key*, by calling *test* with two arguments: the element or its CAR, and *key*. The arguments are passed in that order so that you can get useful results using `string-match` with an alist that contains regular expressions (see Section 35.4 [Regexp Search], page 1011). If *test* is omitted or `nil`, `equal` is used for comparison.

If an alist element matches *key* by this criterion, then `assoc-default` returns a value based on this element. If the element is a cons, then the value is the element's CDR. Otherwise, the return value is *default*.

If no alist element matches *key*, `assoc-default` returns `nil`.

`copy-alist` *alist* [Function]

This function returns a two-level deep copy of *alist*: it creates a new copy of each association, so that you can alter the associations of the new alist without changing the old one.

```
(setq needles-per-cluster
      '((2 . ("Austrian Pine" "Red Pine"))
        (3 . ("Pitch Pine"))
        (5 . ("White Pine"))))
⇒
((2 "Austrian Pine" "Red Pine")
```

```
 (3 "Pitch Pine")
 (5 "White Pine"))

(setq copy (copy-alist needles-per-cluster))
⇒
((2 "Austrian Pine" "Red Pine")
 (3 "Pitch Pine")
 (5 "White Pine"))

(eq needles-per-cluster copy)
     ⇒ nil
(equal needles-per-cluster copy)
     ⇒ t
(eq (car needles-per-cluster) (car copy))
     ⇒ nil
(cdr (car (cdr needles-per-cluster)))
     ⇒ ("Pitch Pine")
(eq (cdr (car (cdr needles-per-cluster)))
    (cdr (car (cdr copy))))
     ⇒ t
```

This example shows how `copy-alist` makes it possible to change the associations of one copy without affecting the other:

```
(setcdr (assq 3 copy) '("Martian Vacuum Pine"))
(cdr (assq 3 needles-per-cluster))
     ⇒ ("Pitch Pine")
```

`assq-delete-all` *key alist* [Function]

This function deletes from *alist* all the elements whose CAR is `eq` to *key*, much as if you used `delq` to delete each such element one by one. It returns the shortened alist, and often modifies the original list structure of *alist*. For correct results, use the return value of `assq-delete-all` rather than looking at the saved value of *alist*.

```
(setq alist (list '(foo 1) '(bar 2) '(foo 3) '(lose 4)))
     ⇒ ((foo 1) (bar 2) (foo 3) (lose 4))
(assq-delete-all 'foo alist)
     ⇒ ((bar 2) (lose 4))
alist
     ⇒ ((foo 1) (bar 2) (lose 4))
```

`assoc-delete-all` *key alist* **&optional** *test* [Function]

This function is like `assq-delete-all` except that it accepts an optional argument *test*, a predicate function to compare the keys in *alist*. If omitted or `nil`, *test* defaults to `equal`. As `assq-delete-all`, this function often modifies the original list structure of *alist*.

`rassq-delete-all` *value alist* [Function]

This function deletes from *alist* all the elements whose CDR is `eq` to *value*. It returns the shortened alist, and often modifies the original list structure of *alist*. `rassq-delete-all` is like `assq-delete-all` except that it compares the CDR of each *alist* association instead of the CAR.

`let-alist` *alist body* [Macro]

Creates a binding for each symbol used as keys the association list *alist*, prefixed with dot. This can be useful when accessing several items in the same association list, and it's best understood through a simple example:

```
(setq colors '((rose . red) (lily . white) (buttercup . yellow)))
(let-alist colors
  (if (eq .rose 'red)
      .lily))
     ⇒ white
```

The *body* is inspected at compilation time, and only the symbols that appear in *body* with a '`.`' as the first character in the symbol name will be bound. Finding the keys is done with `assq`, and the `cdr` of the return value of this `assq` is assigned as the value for the binding.

Nested association lists is supported:

```
(setq colors '((rose . red) (lily (belladonna . yellow) (brindisi . pink))))
(let-alist colors
  (if (eq .rose 'red)
      .lily.belladonna))
     ⇒ yellow
```

Nesting `let-alist` inside each other is allowed, but the code in the inner `let-alist` can't access the variables bound by the outer `let-alist`.

5.9 Property Lists

A *property list* (*plist* for short) is a list of paired elements. Each of the pairs associates a property name (usually a symbol) with a property or value. Here is an example of a property list:

```
(pine cones numbers (1 2 3) color "blue")
```

This property list associates `pine` with `cones`, `numbers` with `(1 2 3)`, and `color` with `"blue"`. The property names and values can be any Lisp objects, but the names are usually symbols (as they are in this example).

Property lists are used in several contexts. For instance, the function `put-text-property` takes an argument which is a property list, specifying text properties and associated values which are to be applied to text in a string or buffer. See Section 33.19 [Text Properties], page 919.

Another prominent use of property lists is for storing symbol properties. Every symbol possesses a list of properties, used to record miscellaneous information about the symbol; these properties are stored in the form of a property list. See Section 9.4 [Symbol Properties], page 138.

`plistp` *object* [Function]

This predicate function returns non-`nil` if *object* is a valid property list.

5.9.1 Property Lists and Association Lists

Association lists (see Section 5.8 [Association Lists], page 95) are very similar to property lists. In contrast to association lists, the order of the pairs in the property list is not significant, since the property names must be distinct.

Property lists are better than association lists for attaching information to various Lisp function names or variables. If your program keeps all such information in one association list, it will typically need to search that entire list each time it checks for an association for a particular Lisp function name or variable, which could be slow. By contrast, if you keep the same information in the property lists of the function names or variables themselves, each search will scan only the length of one property list, which is usually short. This is why the documentation for a variable is recorded in a property named `variable-documentation`. The byte compiler likewise uses properties to record those functions needing special treatment.

However, association lists have their own advantages. Depending on your application, it may be faster to add an association to the front of an association list than to update a property. All properties for a symbol are stored in the same property list, so there is a possibility of a conflict between different uses of a property name. (For this reason, it is a good idea to choose property names that are probably unique, such as by beginning the property name with the program's usual name-prefix for variables and functions.) An association list may be used like a stack where associations are pushed on the front of the list and later discarded; this is not possible with a property list.

5.9.2 Property Lists Outside Symbols

The following functions can be used to manipulate property lists. They all default to comparing property names using `eq`.

`plist-get` *plist property* **&optional** *predicate* [Function]

This returns the value of the *property* property stored in the property list *plist*. Comparisons are done with *predicate*, which defaults to `eq`. It accepts a malformed *plist* argument. If *property* is not found in the *plist*, it returns `nil`. For example,

```
(plist-get '(foo 4) 'foo)
     ⇒ 4
(plist-get '(foo 4 bad) 'foo)
     ⇒ 4
(plist-get '(foo 4 bad) 'bad)
     ⇒ nil
(plist-get '(foo 4 bad) 'bar)
     ⇒ nil
```

`plist-put` *plist property value* **&optional** *predicate* [Function]

This stores *value* as the value of the *property* property in the property list *plist*. Comparisons are done with *predicate*, which defaults to `eq`. It may modify *plist* destructively, or it may construct a new list structure without altering the old. The function returns the modified property list, so you can store that back in the place where you got *plist*. For example,

```
(setq my-plist (list 'bar t 'foo 4))
```

```
     ⇒ (bar t foo 4)
(setq my-plist (plist-put my-plist 'foo 69))
     ⇒ (bar t foo 69)
(setq my-plist (plist-put my-plist 'quux '(a)))
     ⇒ (bar t foo 69 quux (a))
```

`lax-plist-get` *plist property* [Function]
This obsolete function is like `plist-get` except that it compares properties using `equal` instead of `eq`.

`lax-plist-put` *plist property value* [Function]
This obsolete function is like `plist-put` except that it compares properties using `equal` instead of `eq`.

`plist-member` *plist property* **&optional** *predicate* [Function]
This returns non-`nil` if *plist* contains the given *property*. Comparisons are done with *predicate*, which defaults to `eq`. Unlike `plist-get`, this allows you to distinguish between a missing property and a property with the value `nil`. The value is actually the tail of *plist* whose `car` is *property*.

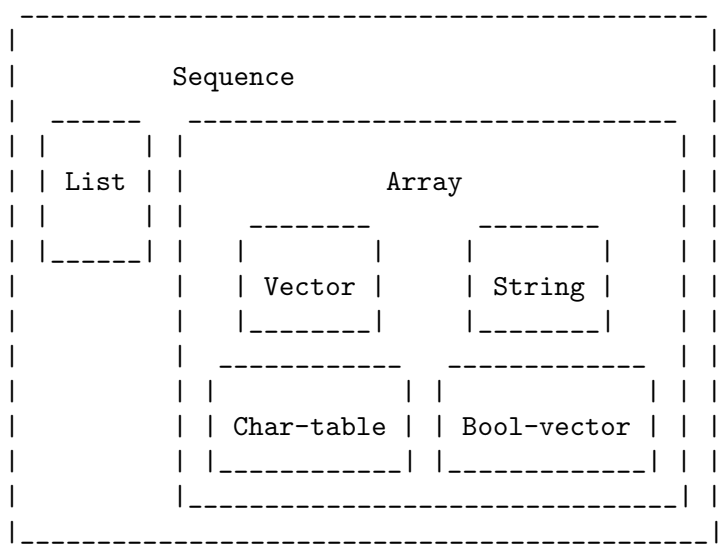
6 Sequences, Arrays, and Vectors

The *sequence* type is the union of two other Lisp types: lists and arrays. In other words, any list is a sequence, and any array is a sequence. The common property that all sequences have is that each is an ordered collection of elements.

An *array* is a fixed-length object with a slot for each of its elements. All the elements are accessible in constant time. The four types of arrays are strings, vectors, char-tables and bool-vectors.

A list is a sequence of elements, but it is not a single primitive object; it is made of cons cells, one cell per element. Finding the *n*th element requires looking through *n* cons cells, so elements farther from the beginning of the list take longer to access. But it is possible to add elements to the list, or remove elements.

The following diagram shows the relationship between these types:



6.1 Sequences

This section describes functions that accept any kind of sequence.

`sequencep` *object* [Function]

This function returns `t` if *object* is a list, vector, string, bool-vector, or char-table, `nil` otherwise. See also `seqp` below.

`length` *sequence* [Function]

This function returns the number of elements in *sequence*. The function signals the `wrong-type-argument` error if the argument is not a sequence or is a dotted list; it signals the `circular-list` error if the argument is a circular list. For a char-table, the value returned is always one more than the maximum Emacs character code.

See [Definition of safe-length], page 81, for the related function `safe-length`.

```
(length '(1 2 3))
    ⇒ 3
```

```
(length ())
    ⇒ 0
(length "foobar")
    ⇒ 6
(length [1 2 3])
    ⇒ 3
(length (make-bool-vector 5 nil))
    ⇒ 5
```

See also `string-bytes`, in Section 34.1 [Text Representations], page 966.

If you need to compute the width of a string on display, you should use `string-width` (see Section 41.10 [Size of Displayed Text], page 1161), not `length`, since `length` only counts the number of characters, but does not account for the display width of each character.

`length<` *sequence length* [Function]
Return non-`nil` if *sequence* is shorter than *length*. This may be more efficient than computing the length of *sequence* if *sequence* is a long list.

`length>` *sequence length* [Function]
Return non-`nil` if *sequence* is longer than *length*.

`length=` *sequence length* [Function]
Return non-`nil` if the length of *sequence* is equal to *length*.

`elt` *sequence index* [Function]
This function returns the element of *sequence* indexed by *index*. Legitimate values of *index* are integers ranging from 0 up to one less than the length of *sequence*. If *sequence* is a list, out-of-range values behave as for `nth`. See [Definition of nth], page 80. Otherwise, out-of-range values trigger an `args-out-of-range` error.

```
(elt [1 2 3 4] 2)
     ⇒ 3
(elt '(1 2 3 4) 2)
     ⇒ 3
;; We use string to show clearly which character elt returns.
(string (elt "1234" 2))
     ⇒ "3"
(elt [1 2 3 4] 4)
     error  Args out of range: [1 2 3 4], 4
(elt [1 2 3 4] -1)
     error  Args out of range: [1 2 3 4], -1
```

This function generalizes `aref` (see Section 6.3 [Array Functions], page 116) and `nth` (see [Definition of nth], page 80).

`copy-sequence` *seqr* [Function]
This function returns a copy of *seqr*, which should be either a sequence or a record. The copy is the same type of object as the original, and it has the same elements in the same order. However, if *seqr* is empty, like a string or a vector of zero length, the value returned by this function might not be a copy, but an empty object of the same type and identical to *seqr*.

Storing a new element into the copy does not affect the original *seqr*, and vice versa. However, the elements of the copy are not copies; they are identical (`eq`) to the elements of the original. Therefore, changes made within these elements, as found via the copy, are also visible in the original.

If the argument is a string with text properties, the property list in the copy is itself a copy, not shared with the original's property list. However, the actual values of the properties are shared. See Section 33.19 [Text Properties], page 919.

This function does not work for dotted lists. Trying to copy a circular list may cause an infinite loop.

See also `append` in Section 5.4 [Building Lists], page 82, `concat` in Section 4.3 [Creating Strings], page 55, and `vconcat` in Section 6.5 [Vector Functions], page 118, for other ways to copy sequences.

```
(setq bar (list 1 2))
     ⇒ (1 2)
(setq x (vector 'foo bar))
     ⇒ [foo (1 2)]
(setq y (copy-sequence x))
     ⇒ [foo (1 2)]

(eq x y)
     ⇒ nil
(equal x y)
     ⇒ t
(eq (elt x 1) (elt y 1))
     ⇒ t

;; Replacing an element of one sequence.
(aset x 0 'quux)
x ⇒ [quux (1 2)]
y ⇒ [foo (1 2)]

;; Modifying the inside of a shared element.
(setcar (aref x 1) 69)
x ⇒ [quux (69 2)]
y ⇒ [foo (69 2)]
```

`reverse` *sequence* [Function]

This function creates a new sequence whose elements are the elements of *sequence*, but in reverse order. The original argument *sequence* is *not* altered. Note that char-tables cannot be reversed.

```
(setq x '(1 2 3 4))
     ⇒ (1 2 3 4)
(reverse x)
     ⇒ (4 3 2 1)
x
     ⇒ (1 2 3 4)
```

```
(setq x [1 2 3 4])
     ⇒ [1 2 3 4]
(reverse x)
     ⇒ [4 3 2 1]
x
     ⇒ [1 2 3 4]
(setq x "xyzzy")
     ⇒ "xyzzy"
(reverse x)
     ⇒ "yzzyx"
x
     ⇒ "xyzzy"
```

`nreverse` *sequence* [Function]

This function reverses the order of the elements of *sequence*. Unlike `reverse` the original *sequence* may be modified.

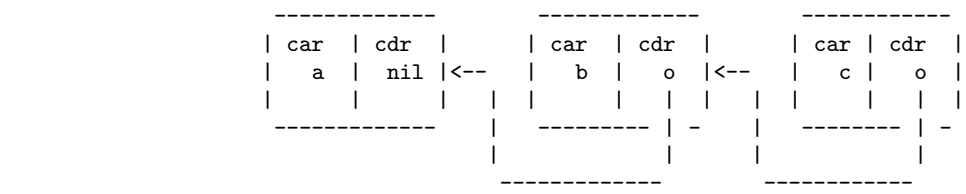
For example:

```
(setq x (list 'a 'b 'c))
     ⇒ (a b c)
x
     ⇒ (a b c)
(nreverse x)
     ⇒ (c b a)
;; The cons cell that was first is now last.
x
     ⇒ (a)
```

To avoid confusion, we usually store the result of `nreverse` back in the same variable which held the original list:

```
(setq x (nreverse x))
```

Here is the `nreverse` of our favorite example, `(a b c)`, presented graphically:



For the vector, it is even simpler because you don't need setq:

```
(setq x (copy-sequence [1 2 3 4]))
     ⇒ [1 2 3 4]
(nreverse x)
     ⇒ [4 3 2 1]
x
     ⇒ [4 3 2 1]
```

Note that unlike `reverse`, this function doesn't work with strings. Although you can alter string data by using `aset`, it is strongly encouraged to treat strings as immutable even when they are mutable. See Section 2.9 [Mutability], page 37.

`sort` *sequence predicate* [Function]

This function sorts *sequence* stably. Note that this function doesn't work for all sequences; it may be used only for lists and vectors. If *sequence* is a list, it is modified destructively. This functions returns the sorted *sequence* and compares elements using *predicate*. A stable sort is one in which elements with equal sort keys maintain their relative order before and after the sort. Stability is important when successive sorts are used to order elements according to different criteria.

The argument *predicate* must be a function that accepts two arguments. It is called with two elements of *sequence*. To get an increasing order sort, the *predicate* should return non-`nil` if the first element is "less" than the second, or `nil` if not.

The comparison function *predicate* must give reliable results for any given pair of arguments, at least within a single call to `sort`. It must be *antisymmetric*; that is, if *a* is less than *b*, *b* must not be less than *a*. It must be *transitive*—that is, if *a* is less than *b*, and *b* is less than *c*, then *a* must be less than *c*. If you use a comparison function which does not meet these requirements, the result of `sort` is unpredictable.

The destructive aspect of `sort` for lists is that it reuses the cons cells forming *sequence* by changing their contents, possibly rearranging them in a different order. This means that the value of the input list is undefined after sorting; only the list returned by `sort` has a well-defined value. Example:

```
(setq nums (list 2 1 4 3 0))
(sort nums #'<)
     ⇒ (0 1 2 3 4)
     ; nums is unpredictable at this point
```

Most often we store the result back into the variable that held the original list:

```
(setq nums (sort nums #'<))
```

If you wish to make a sorted copy without destroying the original, copy it first and then sort:

```
(setq nums (list 2 1 4 3 0))
(sort (copy-sequence nums) #'<)
     ⇒ (0 1 2 3 4)
nums
     ⇒ (2 1 4 3 0)
```

For the better understanding of what stable sort is, consider the following vector example. After sorting, all items whose `car` is 8 are grouped at the beginning of `vector`, but their relative order is preserved. All items whose `car` is 9 are grouped at the end of `vector`, but their relative order is also preserved:

```
(setq
  vector
  (vector '(8 . "xxx") '(9 . "aaa") '(8 . "bbb") '(9 . "zzz")
          '(9 . "ppp") '(8 . "ttt") '(8 . "eee") '(9 . "fff")))
     ⇒ [(8 . "xxx") (9 . "aaa") (8 . "bbb") (9 . "zzz")
        (9 . "ppp") (8 . "ttt") (8 . "eee") (9 . "fff")]
```

```
(sort vector (lambda (x y) (< (car x) (car y))))
     ⇒ [(8 . "xxx") (8 . "bbb") (8 . "ttt") (8 . "eee")
        (9 . "aaa") (9 . "zzz") (9 . "ppp") (9 . "fff")]
```

See Section 33.15 [Sorting], page 908, for more functions that perform sorting. See `documentation` in Section 25.2 [Accessing Documentation], page 598, for a useful example of `sort`.

The `seq.el` library provides the following additional sequence manipulation macros and functions, prefixed with `seq-`.

All functions defined in this library are free of side-effects; i.e., they do not modify any sequence (list, vector, or string) that you pass as an argument. Unless otherwise stated, the result is a sequence of the same type as the input. For those functions that take a predicate, this should be a function of one argument.

The `seq.el` library can be extended to work with additional types of sequential data-structures. For that purpose, all functions are defined using `cl-defgeneric`. See Section 13.8 [Generic Functions], page 245, for more details about using `cl-defgeneric` for adding extensions.

`seq-elt` *sequence index* [Function]

This function returns the element of *sequence* at the specified *index*, which is an integer whose valid value range is zero to one less than the length of *sequence*. For out-of-range values on built-in sequence types, `seq-elt` behaves like `elt`. For the details, see [Definition of elt], page 103.

```
(seq-elt [1 2 3 4] 2)
⇒ 3
```

`seq-elt` returns places settable using `setf` (see Section 12.17.1 [Setting Generalized Variables], page 224).

```
(setq vec [1 2 3 4])
(setf (seq-elt vec 2) 5)
vec
⇒ [1 2 5 4]
```

`seq-length` *sequence* [Function]

This function returns the number of elements in *sequence*. For built-in sequence types, `seq-length` behaves like `length`. See [Definition of length], page 102.

`seqp` *object* [Function]

This function returns non-`nil` if *object* is a sequence (a list or array), or any additional type of sequence defined via `seq.el` generic functions. This is an extensible variant of `sequencep`.

```
(seqp [1 2])
⇒ t
(seqp 2)
⇒ nil
```

`seq-drop` *sequence n* [Function]

This function returns all but the first *n* (an integer) elements of *sequence*. If *n* is negative or zero, the result is *sequence*.

```
(seq-drop [1 2 3 4 5 6] 3)
⇒ [4 5 6]
(seq-drop "hello world" -4)
⇒ "hello world"
```

`seq-take` *sequence n* [Function]

This function returns the first *n* (an integer) elements of *sequence*. If *n* is negative or zero, the result is `nil`.

```
(seq-take '(1 2 3 4) 3)
⇒ (1 2 3)
(seq-take [1 2 3 4] 0)
⇒ []
```

`seq-take-while` *predicate sequence* [Function]

This function returns the members of *sequence* in order, stopping before the first one for which *predicate* returns `nil`.

```
(seq-take-while (lambda (elt) (> elt 0)) '(1 2 3 -1 -2))
⇒ (1 2 3)
(seq-take-while (lambda (elt) (> elt 0)) [-1 4 6])
⇒ []
```

`seq-drop-while` *predicate sequence* [Function]

This function returns the members of *sequence* in order, starting from the first one for which *predicate* returns `nil`.

```
(seq-drop-while (lambda (elt) (> elt 0)) '(1 2 3 -1 -2))
⇒ (-1 -2)
(seq-drop-while (lambda (elt) (< elt 0)) [1 4 6])
⇒ [1 4 6]
```

`seq-split` *sequence length* [Function]

This function returns a list consisting of sub-sequences of *sequence* of (at most) length *length*. (The final element may be shorter than *length* if the length of *sequence* isn't a multiple of *length*.

```
(seq-split [0 1 2 3 4] 2)
⇒ ([0 1] [2 3] [4])
```

`seq-do` *function sequence* [Function]

This function applies *function* to each element of *sequence* in turn (presumably for side effects), and returns *sequence*.

`seq-map` *function sequence* [Function]

This function returns the result of applying *function* to each element of *sequence*. The returned value is a list.

```
(seq-map #'1+ '(2 4 6))
⇒ (3 5 7)
(seq-map #'symbol-name [foo bar])
⇒ ("foo" "bar")
```

`seq-map-indexed` *function sequence* [Function]

This function returns the result of applying *function* to each element of *sequence* and its index within *seq*. The returned value is a list.

```
(seq-map-indexed (lambda (elt idx)
                   (list idx elt))
                 '(a b c))
⇒ ((0 a) (1 b) (2 c))
```

`seq-mapn` *function* **&rest** *sequences* [Function]

This function returns the result of applying *function* to each element of *sequences*. The arity (see Section 13.1 [What Is a Function], page 230) of *function* must match the number of sequences. Mapping stops at the end of the shortest sequence, and the returned value is a list.

```
(seq-mapn #'+ '(2 4 6) '(20 40 60))
⇒ (22 44 66)
(seq-mapn #'concat '("moskito" "bite") ["bee" "sting"])
⇒ ("moskitobee" "bitesting")
```

`seq-filter` *predicate sequence* [Function]

This function returns a list of all the elements in *sequence* for which *predicate* returns non-`nil`.

```
(seq-filter (lambda (elt) (> elt 0)) [1 -1 3 -3 5])
⇒ (1 3 5)
(seq-filter (lambda (elt) (> elt 0)) '(-1 -3 -5))
⇒ nil
```

`seq-remove` *predicate sequence* [Function]

This function returns a list of all the elements in *sequence* for which *predicate* returns `nil`.

```
(seq-remove (lambda (elt) (> elt 0)) [1 -1 3 -3 5])
⇒ (-1 -3)
(seq-remove (lambda (elt) (< elt 0)) '(-1 -3 -5))
⇒ nil
```

`seq-remove-at-position` *sequence n* [Function]

This function returns a copy of *sequence* where the element at (zero-based) index *n* got removed. The result is a sequence of the same type as *sequence*.

```
(seq-remove-at-position [1 -1 3 -3 5] 0)
⇒ [-1 3 -3 5]
(seq-remove-at-position [1 -1 3 -3 5] 3)
⇒ [1 -1 3 5]
```

`seq-keep` *function sequence* [Function]

This function returns a list of all non-`nil` results from calling *function* on the elements in *sequence*.

```
(seq-keep #'cl-digit-char-p '(?6 ?a ?7))
⇒ (6 7)
```

`seq-reduce` *function sequence initial-value* [Function]

This function returns the result of calling *function* with *initial-value* and the first element of *sequence*, then calling *function* with that result and the second element of *sequence*, then with that result and the third element of *sequence*, etc. *function* should be a function of two arguments.

function is called with two arguments. *initial-value* (and then the accumulated value) is used as the first argument, and the elements in *sequence* are used for the second argument.

If *sequence* is empty, this returns *initial-value* without calling *function*.

```
(seq-reduce #'+ [1 2 3 4] 0)
⇒ 10
(seq-reduce #'+ '(1 2 3 4) 5)
⇒ 15
(seq-reduce #'+ '() 3)
⇒ 3
```

`seq-some` *predicate sequence* [Function]

This function returns the first non-`nil` value returned by applying *predicate* to each element of *sequence* in turn.

```
(seq-some #'numberp ["abc" 1 nil])
⇒ t
(seq-some #'numberp ["abc" "def"])
⇒ nil
(seq-some #'null ["abc" 1 nil])
⇒ t
(seq-some #'1+ [2 4 6])
⇒ 3
```

`seq-find` *predicate sequence* **&optional** *default* [Function]

This function returns the first element in *sequence* for which *predicate* returns non-`nil`. If no element matches *predicate*, the function returns *default*.

Note that this function has an ambiguity if the found element is identical to *default*, as in that case it cannot be known whether an element was found or not.

```
(seq-find #'numberp ["abc" 1 nil])
⇒ 1
(seq-find #'numberp ["abc" "def"])
⇒ nil
```

`seq-every-p` *predicate sequence* [Function]

This function returns non-`nil` if applying *predicate* to every element of *sequence* returns non-`nil`.

```
(seq-every-p #'numberp [2 4 6])
⇒ t
(seq-every-p #'numberp [2 4 "6"])
⇒ nil
```

`seq-empty-p` *sequence* [Function]

This function returns non-`nil` if *sequence* is empty.

```
(seq-empty-p "not empty")
⇒ nil
(seq-empty-p "")
⇒ t
```

`seq-count` *predicate sequence* [Function]

This function returns the number of elements in *sequence* for which *predicate* returns non-`nil`.

```
(seq-count (lambda (elt) (> elt 0)) [-1 2 0 3 -2])
⇒ 2
```

`seq-sort` *function sequence* [Function]

This function returns a copy of *sequence* that is sorted according to *function*, a function of two arguments that returns non-`nil` if the first argument should sort before the second.

`seq-sort-by` *function predicate sequence* [Function]

This function is similar to `seq-sort`, but the elements of *sequence* are transformed by applying *function* on them before being sorted. *function* is a function of one argument.

```
(seq-sort-by #'seq-length #'> ["a" "ab" "abc"])
⇒ ["abc" "ab" "a"]
```

`seq-contains-p` *sequence elt* **&optional** *function* [Function]

This function returns non-`nil` if at least one element in *sequence* is equal to *elt*. If the optional argument *function* is non-`nil`, it is a function of two arguments to use instead of the default `equal`.

```
(seq-contains-p '(symbol1 symbol2) 'symbol1)
⇒ t
(seq-contains-p '(symbol1 symbol2) 'symbol3)
⇒ nil
```

`seq-set-equal-p` *sequence1 sequence2* **&optional** *testfn* [Function]

This function checks whether *sequence1* and *sequence2* contain the same elements, regardless of the order. If the optional argument *testfn* is non-`nil`, it is a function of two arguments to use instead of the default `equal`.

```
(seq-set-equal-p '(a b c) '(c b a))
⇒ t
(seq-set-equal-p '(a b c) '(c b))
⇒ nil
(seq-set-equal-p '("a" "b" "c") '("c" "b" "a"))
⇒ t
(seq-set-equal-p '("a" "b" "c") '("c" "b" "a") #'eq)
⇒ nil
```

`seq-position` *sequence elt* **&optional** *function* [Function]

This function returns the (zero-based) index of the first element in *sequence* that is equal to *elt*. If the optional argument *function* is non-`nil`, it is a function of two arguments to use instead of the default `equal`.

```
(seq-position '(a b c) 'b)
⇒ 1
(seq-position '(a b c) 'd)
⇒ nil
```

`seq-positions` *sequence elt* **&optional** *testfn* [Function]

This function returns a list of the (zero-based) indices of the elements in *sequence* for which *testfn* returns non-`nil` when passed the element and *elt* as arguments. *testfn* defaults to `equal`.

```
(seq-positions '(a b c a d) 'a)
⇒ (0 3)
(seq-positions '(a b c a d) 'z)
⇒ nil
(seq-positions '(11 5 7 12 9 15) 10 #'>=)
⇒ (0 3 5)
```

`seq-uniq` *sequence* **&optional** *function* [Function]

This function returns a list of the elements of *sequence* with duplicates removed. If the optional argument *function* is non-`nil`, it is a function of two arguments to use instead of the default `equal`.

```
(seq-uniq '(1 2 2 1 3))
⇒ (1 2 3)
(seq-uniq '(1 2 2.0 1.0) #'=)
⇒ (1 2)
```

`seq-subseq` *sequence start* **&optional** *end* [Function]

This function returns a subset of *sequence* from *start* to *end*, both integers (*end* defaults to the last element). If *start* or *end* is negative, it counts from the end of *sequence*.

```
(seq-subseq '(1 2 3 4 5) 1)
⇒ (2 3 4 5)
(seq-subseq '[1 2 3 4 5] 1 3)
⇒ [2 3]
(seq-subseq '[1 2 3 4 5] -3 -1)
⇒ [3 4]
```

`seq-concatenate` *type* **&rest** *sequences* [Function]

This function returns a sequence of type *type* made of the concatenation of *sequences*. *type* may be: `vector`, `list` or `string`.

```
(seq-concatenate 'list '(1 2) '(3 4) [5 6])
⇒ (1 2 3 4 5 6)
(seq-concatenate 'string "Hello " "world")
⇒ "Hello world"
```

`seq-mapcat` *function sequence* **&optional** *type* [Function]
This function returns the result of applying `seq-concatenate` to the result of applying *function* to each element of *sequence*. The result is a sequence of type *type*, or a list if *type* is `nil`.

```
(seq-mapcat #'seq-reverse '((3 2 1) (6 5 4)))
⇒ (1 2 3 4 5 6)
```

`seq-partition` *sequence n* [Function]
This function returns a list of the elements of *sequence* grouped into sub-sequences of length *n*. The last sequence may contain less elements than *n*. *n* must be an integer. If *n* is a negative integer or 0, the return value is `nil`.

```
(seq-partition '(0 1 2 3 4 5 6 7) 3)
⇒ ((0 1 2) (3 4 5) (6 7))
```

`seq-union` *sequence1 sequence2* **&optional** *function* [Function]
This function returns a list of the elements that appear either in *sequence1* or *sequence2*. The elements of the returned list are all unique, in the sense that no two elements there will compare equal. If the optional argument *function* is non-`nil`, it should be a function of two arguments to use to compare elements, instead of the default `equal`.

```
(seq-union [1 2 3] [3 5])
⇒ (1 2 3 5)
```

`seq-intersection` *sequence1 sequence2* **&optional** *function* [Function]
This function returns a list of the elements that appear both in *sequence1* and *sequence2*. If the optional argument *function* is non-`nil`, it is a function of two arguments to use to compare elements instead of the default `equal`.

```
(seq-intersection [2 3 4 5] [1 3 5 6 7])
⇒ (3 5)
```

`seq-difference` *sequence1 sequence2* **&optional** *function* [Function]
This function returns a list of the elements that appear in *sequence1* but not in *sequence2*. If the optional argument *function* is non-`nil`, it is a function of two arguments to use to compare elements instead of the default `equal`.

```
(seq-difference '(2 3 4 5) [1 3 5 6 7])
⇒ (2 4)
```

`seq-group-by` *function sequence* [Function]
This function separates the elements of *sequence* into an alist whose keys are the result of applying *function* to each element of *sequence*. Keys are compared using `equal`.

```
(seq-group-by #'integerp '(1 2.1 3 2 3.2))
⇒ ((t 1 3 2) (nil 2.1 3.2))
(seq-group-by #'car '((a 1) (b 2) (a 3) (c 4)))
⇒ ((b (b 2)) (a (a 1) (a 3)) (c (c 4)))
```

`seq-into` *sequence type* [Function]

This function converts the sequence *sequence* into a sequence of type *type*. *type* can be one of the following symbols: `vector`, `string` or `list`.

```
(seq-into [1 2 3] 'list)
⇒ (1 2 3)
(seq-into nil 'vector)
⇒ []
(seq-into "hello" 'vector)
⇒ [104 101 108 108 111]
```

`seq-min` *sequence* [Function]

This function returns the smallest element of *sequence*. The elements of *sequence* must be numbers or markers (see Chapter 32 [Markers], page 871).

```
(seq-min [3 1 2])
⇒ 1
(seq-min "Hello")
⇒ 72
```

`seq-max` *sequence* [Function]

This function returns the largest element of *sequence*. The elements of *sequence* must be numbers or markers.

```
(seq-max [1 3 2])
⇒ 3
(seq-max "Hello")
⇒ 111
```

`seq-doseq` (*var sequence*) *body*. . . [Macro]

This macro is like `dolist` (see Section 11.5 [Iteration], page 174), except that *sequence* can be a list, vector or string. This is primarily useful for side-effects.

`seq-let` *var-sequence val-sequence body*. . . [Macro]

This macro binds the variables defined in *var-sequence* to the values that are the corresponding elements of *val-sequence*. This is known as *destructuring binding*. The elements of *var-sequence* can themselves include sequences, allowing for nested destructuring.

The *var-sequence* sequence can also include the `&rest` marker followed by a variable name to be bound to the rest of *val-sequence*.

```
(seq-let [first second] [1 2 3 4]
  (list first second))
⇒ (1 2)
(seq-let (_ a _ b) '(1 2 3 4)
  (list a b))
⇒ (2 4)
(seq-let [a [b [c]]] [1 [2 [3]]]
  (list a b c))
⇒ (1 2 3)
(seq-let [a b &rest others] [1 2 3 4]
  others)
```

```
⇒ [3 4]
```

The `pcase` patterns provide an alternative facility for destructuring binding, see Section 11.4.4 [Destructuring with pcase Patterns], page 172.

`seq-setq` *var-sequence val-sequence* [Macro]

This macro works similarly to `seq-let`, except that values are assigned to variables as if by `setq` instead of as in a `let` binding.

```
(let ((a nil)
      (b nil))
  (seq-setq (_ a _ b) '(1 2 3 4))
  (list a b))
⇒ (2 4)
```

`seq-random-elt` *sequence* [Function]

This function returns an element of *sequence* taken at random.

```
(seq-random-elt [1 2 3 4])
⇒ 3
(seq-random-elt [1 2 3 4])
⇒ 2
(seq-random-elt [1 2 3 4])
⇒ 4
(seq-random-elt [1 2 3 4])
⇒ 2
(seq-random-elt [1 2 3 4])
⇒ 1
```

If *sequence* is empty, this function signals an error.

6.2 Arrays

An *array* object has slots that hold a number of other Lisp objects, called the elements of the array. Any element of an array may be accessed in constant time. In contrast, the time to access an element of a list is proportional to the position of that element in the list.

Emacs defines four types of array, all one-dimensional: *strings* (see Section 2.4.8 [String Type], page 20), *vectors* (see Section 2.4.9 [Vector Type], page 22), *bool-vectors* (see Section 2.4.11 [Bool-Vector Type], page 23), and *char-tables* (see Section 2.4.10 [Char-Table Type], page 22). Vectors and char-tables can hold elements of any type, but strings can only hold characters, and bool-vectors can only hold `t` and `nil`.

All four kinds of array share these characteristics:

- The first element of an array has index zero, the second element has index 1, and so on. This is called *zero-origin* indexing. For example, an array of four elements has indices 0, 1, 2, and 3.
- The length of the array is fixed once you create it; you cannot change the length of an existing array.
- For purposes of evaluation, the array is a constant—i.e., it evaluates to itself.
- The elements of an array may be referenced or changed with the functions `aref` and `aset`, respectively (see Section 6.3 [Array Functions], page 116).

When you create an array, other than a char-table, you must specify its length. You cannot specify the length of a char-table, because that is determined by the range of character codes.

In principle, if you want an array of text characters, you could use either a string or a vector. In practice, we always choose strings for such applications, for four reasons:

- They occupy one-fourth the space of a vector of the same elements.
- Strings are printed in a way that shows the contents more clearly as text.
- Strings can hold text properties. See Section 33.19 [Text Properties], page 919.
- Many of the specialized editing and I/O facilities of Emacs accept only strings. For example, you cannot insert a vector of characters into a buffer the way you can insert a string. See Chapter 4 [Strings and Characters], page 54.

By contrast, for an array of keyboard input characters (such as a key sequence), a vector may be necessary, because many keyboard input characters are outside the range that will fit in a string. See Section 22.8.1 [Key Sequence Input], page 462.

6.3 Functions that Operate on Arrays

In this section, we describe the functions that accept all types of arrays.

`arrayp` *object* [Function]

This function returns `t` if *object* is an array (i.e., a vector, a string, a bool-vector or a char-table).

```
(arrayp [a])
     ⇒ t
(arrayp "asdf")
     ⇒ t
(arrayp (syntax-table))     ;; A char-table.
     ⇒ t
```

`aref` *arr index* [Function]

This function returns the *index*th element of the array or record *arr*. The first element is at index zero.

```
(setq primes [2 3 5 7 11 13])
     ⇒ [2 3 5 7 11 13]
(aref primes 4)
     ⇒ 11
(aref "abcdefg" 1)
     ⇒ 98           ; 'b' is ASCII code 98.
```

See also the function `elt`, in Section 6.1 [Sequence Functions], page 102.

`aset` *array index object* [Function]

This function sets the *index*th element of *array* to be *object*. It returns *object*.

```
(setq w (vector 'foo 'bar 'baz))
     ⇒ [foo bar baz]
(aset w 0 'fu)
     ⇒ fu
w
     ⇒ [fu bar baz]

;; copy-sequence copies the string to be modified later.
(setq x (copy-sequence "asdfasfd"))
     ⇒ "asdfasfd"
(aset x 3 ?Z)
     ⇒ 90
x
     ⇒ "asdZasfd"
```

The *array* should be mutable. See Section 2.9 [Mutability], page 37.

If *array* is a string and *object* is not a character, a `wrong-type-argument` error results. The function converts a unibyte string to multibyte if necessary to insert a character.

`fillarray` *array object* [Function]

This function fills the array *array* with *object*, so that each element of *array* is *object*. It returns *array*.

```
(setq a (copy-sequence [a b c d e f g]))
     ⇒ [a b c d e f g]
(fillarray a 0)
     ⇒ [0 0 0 0 0 0 0]
a
     ⇒ [0 0 0 0 0 0 0]
(setq s (copy-sequence "When in the course"))
     ⇒ "When in the course"
(fillarray s ?-)
     ⇒ "------------------"
```

If *array* is a string and *object* is not a character, a `wrong-type-argument` error results.

The general sequence functions `copy-sequence` and `length` are often useful for objects known to be arrays. See Section 6.1 [Sequence Functions], page 102.

6.4 Vectors

A *vector* is a general-purpose array whose elements can be any Lisp objects. (By contrast, the elements of a string can only be characters. See Chapter 4 [Strings and Characters], page 54.) Vectors are used in Emacs for many purposes: as key sequences (see Section 23.1 [Key Sequences], page 481), as symbol-lookup tables (see Section 9.3 [Creating Symbols], page 135), as part of the representation of a byte-compiled function (see Chapter 17 [Byte Compilation], page 317), and more.

Like other arrays, vectors use zero-origin indexing: the first element has index 0.

Vectors are printed with square brackets surrounding the elements. Thus, a vector whose elements are the symbols `a`, `b` and `a` is printed as `[a b a]`. You can write vectors in the same way in Lisp input.

A vector, like a string or a number, is considered a constant for evaluation: the result of evaluating it is the same vector. This does not evaluate or even examine the elements of the vector. See Section 10.1.1 [Self-Evaluating Forms], page 146. Vectors written with square brackets should not be modified via `aset` or other destructive operations. See Section 2.9 [Mutability], page 37.

Here are examples illustrating these principles:

```
(setq avector [1 two '(three) "four" [five]])
     ⇒ [1 two '(three) "four" [five]]
(eval avector)
     ⇒ [1 two '(three) "four" [five]]
(eq avector (eval avector))
     ⇒ t
```

6.5 Functions for Vectors

Here are some functions that relate to vectors:

`vectorp` *object* [Function]

This function returns `t` if *object* is a vector.

```
(vectorp [a])
     ⇒ t
(vectorp "asdf")
     ⇒ nil
```

`vector` **&rest** *objects* [Function]

This function creates and returns a vector whose elements are the arguments, *objects*.

```
(vector 'foo 23 [bar baz] "rats")
     ⇒ [foo 23 [bar baz] "rats"]
(vector)
     ⇒ []
```

`make-vector` *length object* [Function]

This function returns a new vector consisting of *length* elements, each initialized to *object*.

```
(setq sleepy (make-vector 9 'Z))
     ⇒ [Z Z Z Z Z Z Z Z Z]
```

`vconcat` **&rest** *sequences* [Function]

This function returns a new vector containing all the elements of *sequences*. The arguments *sequences* may be proper lists, vectors, strings or bool-vectors. If no *sequences* are given, the empty vector is returned.

The value is either the empty vector, or is a newly constructed nonempty vector that is not `eq` to any existing vector.

```
(setq a (vconcat '(A B C) '(D E F)))
     ⇒ [A B C D E F]
(eq a (vconcat a))
     ⇒ nil
(vconcat)
     ⇒ []
(vconcat [A B C] "aa" '(foo (6 7)))
     ⇒ [A B C 97 97 foo (6 7)]
```

The `vconcat` function also allows byte-code function objects as arguments. This is a special feature to make it easy to access the entire contents of a byte-code function object. See Section 17.7 [Byte-Code Objects], page 324.

For other concatenation functions, see `mapconcat` in Section 13.6 [Mapping Functions], page 241, `concat` in Section 4.3 [Creating Strings], page 55, and `append` in Section 5.4 [Building Lists], page 82.

The `append` function also provides a way to convert a vector into a list with the same elements:

```
(setq avector [1 two (quote (three)) "four" [five]])
     ⇒ [1 two '(three) "four" [five]]
(append avector nil)
     ⇒ (1 two '(three) "four" [five])
```

6.6 Char-Tables

A char-table is much like a vector, except that it is indexed by character codes. Any valid character code, without modifiers, can be used as an index in a char-table. You can access a char-table's elements with `aref` and `aset`, as with any array. In addition, a char-table can have *extra slots* to hold additional data not associated with particular character codes. Like vectors, char-tables are constants when evaluated, and can hold elements of any type.

Each char-table has a *subtype*, a symbol, which serves two purposes:

- The subtype provides an easy way to tell what the char-table is for. For instance, display tables are char-tables with `display-table` as the subtype, and syntax tables are char-tables with `syntax-table` as the subtype. The subtype can be queried using the function `char-table-subtype`, described below.
- The subtype controls the number of *extra slots* in the char-table. This number is specified by the subtype's `char-table-extra-slots` symbol property (see Section 9.4 [Symbol Properties], page 138), whose value should be an integer between 0 and 10. If the subtype has no such symbol property, the char-table has no extra slots.

A char-table can have a *parent*, which is another char-table. If it does, then whenever the char-table specifies `nil` for a particular character *c*, it inherits the value specified in the parent. In other words, `(aref` *char-table c*`)` returns the value from the parent of *char-table* if *char-table* itself specifies `nil`.

A char-table can also have a *default value*. If so, then `(aref` *char-table c*`)` returns the default value whenever the char-table does not specify any other non-`nil` value.

`make-char-table` *subtype* **&optional** *init* [Function]
Return a newly-created char-table, with subtype *subtype* (a symbol). Each element is initialized to *init*, which defaults to `nil`. You cannot alter the subtype of a char-table after the char-table is created.

There is no argument to specify the length of the char-table, because all char-tables have room for any valid character code as an index.

If *subtype* has the `char-table-extra-slots` symbol property, that specifies the number of extra slots in the char-table. This should be an integer between 0 and 10; otherwise, `make-char-table` raises an error. If *subtype* has no `char-table-extra-slots` symbol property (see Section 5.9 [Property Lists], page 99), the char-table has no extra slots.

`char-table-p` *object* [Function]
This function returns `t` if *object* is a char-table, and `nil` otherwise.

`char-table-subtype` *char-table* [Function]
This function returns the subtype symbol of *char-table*.

There is no special function to access default values in a char-table. To do that, use `char-table-range` (see below).

`char-table-parent` *char-table* [Function]
This function returns the parent of *char-table*. The parent is always either `nil` or another char-table.

`set-char-table-parent` *char-table new-parent* [Function]
This function sets the parent of *char-table* to *new-parent*.

`char-table-extra-slot` *char-table n* [Function]
This function returns the contents of extra slot *n* (zero based) of *char-table*. The number of extra slots in a char-table is determined by its subtype.

`set-char-table-extra-slot` *char-table n value* [Function]
This function stores *value* in extra slot *n* (zero based) of *char-table*.

A char-table can specify an element value for a single character code; it can also specify a value for an entire character set.

`char-table-range` *char-table range* [Function]
This returns the value specified in *char-table* for a range of characters *range*. Here are the possibilities for *range*:

`nil` Refers to the default value.

char Refers to the element for character *char* (supposing *char* is a valid character code).

`(`*from* `.` *to*`)`
A cons cell refers to all the characters in the inclusive range '`[`*from*`..`*to*`]`'. In this case, the function returns the value for the character specified by *from*.

`set-char-table-range` *char-table range value* [Function]

This function sets the value in *char-table* for a range of characters *range*. Here are the possibilities for *range*:

`nil` Refers to the default value.

`t` Refers to the whole range of character codes.

char Refers to the element for character *char* (supposing *char* is a valid character code).

`(`*from* `.` *to*`)` A cons cell refers to all the characters in the inclusive range '`[`*from*`..`*to*`]`'.

`map-char-table` *function char-table* [Function]

This function calls its argument *function* for each element of *char-table* that has a non-`nil` value. The call to *function* is with two arguments, a key and a value. The key is a possible *range* argument for `char-table-range`—either a valid character or a cons cell `(`*from* `.` *to*`)`, specifying a range of characters that share the same value. The value is what `(char-table-range` *char-table key*`)` returns.

Overall, the key-value pairs passed to *function* describe all the values stored in *char-table*.

The return value is always `nil`; to make calls to `map-char-table` useful, *function* should have side effects. For example, here is how to examine the elements of the syntax table:

```
(let (accumulator)
   (map-char-table
    (lambda (key value)
      (setq accumulator
            (cons (list
                   (if (consp key)
                       (list (car key) (cdr key))
                     key)
                   value)
                  accumulator)))
    (syntax-table))
   accumulator)
⇒
(((2597602 4194303) (2)) ((2597523 2597601) (3))
 ... (65379 (5 . 65378)) (65378 (4 . 65379)) (65377 (1))
 ... (12 (0)) (11 (3)) (10 (12)) (9 (0)) ((0 8) (3)))
```

6.7 Bool-vectors

A bool-vector is much like a vector, except that it stores only the values `t` and `nil`. If you try to store any non-`nil` value into an element of the bool-vector, the effect is to store `t` there. As with all arrays, bool-vector indices start from 0, and the length cannot be changed once the bool-vector is created. Bool-vectors are constants when evaluated.

Several functions work specifically with bool-vectors; aside from that, you manipulate them with same functions used for other kinds of arrays.

`make-bool-vector` *length initial* [Function]
Return a new bool-vector of *length* elements, each one initialized to *initial*.

`bool-vector` **&rest** *objects* [Function]
This function creates and returns a bool-vector whose elements are the arguments, *objects*.

`bool-vector-p` *object* [Function]
This returns `t` if *object* is a bool-vector, and `nil` otherwise.

There are also some bool-vector set operation functions, described below:

`bool-vector-exclusive-or` *a b* **&optional** *c* [Function]
Return *bitwise exclusive or* of bool vectors *a* and *b*. If optional argument *c* is given, the result of this operation is stored into *c*. All arguments should be bool vectors of the same length.

`bool-vector-union` *a b* **&optional** *c* [Function]
Return *bitwise or* of bool vectors *a* and *b*. If optional argument *c* is given, the result of this operation is stored into *c*. All arguments should be bool vectors of the same length.

`bool-vector-intersection` *a b* **&optional** *c* [Function]
Return *bitwise and* of bool vectors *a* and *b*. If optional argument *c* is given, the result of this operation is stored into *c*. All arguments should be bool vectors of the same length.

`bool-vector-set-difference` *a b* **&optional** *c* [Function]
Return *set difference* of bool vectors *a* and *b*. If optional argument *c* is given, the result of this operation is stored into *c*. All arguments should be bool vectors of the same length.

`bool-vector-not` *a* **&optional** *b* [Function]
Return *set complement* of bool vector *a*. If optional argument *b* is given, the result of this operation is stored into *b*. All arguments should be bool vectors of the same length.

`bool-vector-subsetp` *a b* [Function]
Return `t` if every `t` value in *a* is also `t` in *b*, `nil` otherwise. All arguments should be bool vectors of the same length.

`bool-vector-count-consecutive` *a b i* [Function]
Return the number of consecutive elements in *a* equal *b* starting at *i*. `a` is a bool vector, *b* is `t` or `nil`, and *i* is an index into `a`.

`bool-vector-count-population` *a* [Function]
Return the number of elements that are `t` in bool vector *a*.

The printed form represents up to 8 boolean values as a single character:

```
(bool-vector t nil t nil)
     ⇒ #&4"^E"
(bool-vector)
     ⇒ #&0""
```

You can use `vconcat` to print a bool-vector like other vectors:

```
(vconcat (bool-vector nil t nil t))
     ⇒ [nil t nil t]
```

Here is another example of creating, examining, and updating a bool-vector:

```
(setq bv (make-bool-vector 5 t))
     ⇒ #&5"^_"
(aref bv 1)
     ⇒ t
(aset bv 3 nil)
     ⇒ nil
bv
     ⇒ #&5"^W"
```

These results make sense because the binary codes for control-_ and control-W are 11111 and 10111, respectively.

6.8 Managing a Fixed-Size Ring of Objects

A *ring* is a fixed-size data structure that supports insertion, deletion, rotation, and modulo-indexed reference and traversal. An efficient ring data structure is implemented by the `ring` package. It provides the functions listed in this section.

Note that several rings in Emacs, like the kill ring and the mark ring, are actually implemented as simple lists, *not* using the `ring` package; thus the following functions won't work on them.

`make-ring` *size* [Function]
: This returns a new ring capable of holding *size* objects. *size* should be an integer.

`ring-p` *object* [Function]
: This returns `t` if *object* is a ring, `nil` otherwise.

`ring-size` *ring* [Function]
: This returns the maximum capacity of the *ring*.

`ring-length` *ring* [Function]
: This returns the number of objects that *ring* currently contains. The value will never exceed that returned by `ring-size`.

`ring-elements` *ring* [Function]
: This returns a list of the objects in *ring*, in order, newest first.

`ring-copy` *ring* [Function]
: This returns a new ring which is a copy of *ring*. The new ring contains the same (`eq`) objects as *ring*.

`ring-empty-p` *ring* [Function]
: This returns `t` if *ring* is empty, `nil` otherwise.

The newest element in the ring always has index 0. Higher indices correspond to older elements. Indices are computed modulo the ring length. Index −1 corresponds to the oldest element, −2 to the next-oldest, and so forth.

`ring-ref` *ring index* [Function]
: This returns the object in *ring* found at index *index*. *index* may be negative or greater than the ring length. If *ring* is empty, `ring-ref` signals an error.

`ring-insert` *ring object* [Function]
: This inserts *object* into *ring*, making it the newest element, and returns *object*.

  If the ring is full, insertion removes the oldest element to make room for the new element.

`ring-remove` *ring* **&optional** *index* [Function]
: Remove an object from *ring*, and return that object. The argument *index* specifies which item to remove; if it is `nil`, that means to remove the oldest item. If *ring* is empty, `ring-remove` signals an error.

`ring-insert-at-beginning` *ring object* [Function]
: This inserts *object* into *ring*, treating it as the oldest element. The return value is not significant.

  If the ring is full, this function removes the newest element to make room for the inserted element.

`ring-resize` *ring size* [Function]
: Set the size of *ring* to *size*. If the new size is smaller, then the oldest items in the ring are discarded.

If you are careful not to exceed the ring size, you can use the ring as a first-in-first-out queue. For example:

```
(let ((fifo (make-ring 5)))
  (mapc (lambda (obj) (ring-insert fifo obj))
        '(0 one "two"))
  (list (ring-remove fifo) t
        (ring-remove fifo) t
        (ring-remove fifo)))
     ⇒ (0 t one t "two")
```

7 Records

The purpose of records is to allow programmers to create objects with new types that are not built into Emacs. They are used as the underlying representation of `cl-defstruct` and `defclass` instances.

Internally, a record object is much like a vector; its slots can be accessed using `aref` and it can be copied using `copy-sequence`. However, the first slot is used to hold its type as returned by `type-of`. Also, in the current implementation records can have at most 4096 slots, whereas vectors can be much larger. Like arrays, records use zero-origin indexing: the first slot has index 0.

The type slot should be a symbol or a type descriptor. If it's a type descriptor, the symbol naming its type will be returned; Section 2.4.18 [Type Descriptors], page 25. Any other kind of object is returned as-is.

The printed representation of records is '`#s`' followed by a list specifying the contents. The first list element must be the record type. The following elements are the record slots.

To avoid conflicts with other type names, Lisp programs that define new types of records should normally use the naming conventions of the package where these record types are introduced for the names of the types. Note that the names of the types which could possibly conflict might not be known at the time the package defining a record type is loaded; they could be loaded at some future point in time.

A record is considered a constant for evaluation: the result of evaluating it is the same record. This does not evaluate or even examine the slots. See Section 10.1.1 [Self-Evaluating Forms], page 146.

7.1 Record Functions

`recordp` *object* [Function]

This function returns `t` if *object* is a record.

```
(recordp #s(a))
     ⇒ t
```

`record` *type* **&rest** *objects* [Function]

This function creates and returns a record whose type is *type* and remaining slots are the rest of the arguments, *objects*.

```
(record 'foo 23 [bar baz] "rats")
     ⇒ #s(foo 23 [bar baz] "rats")
```

`make-record` *type length object* [Function]

This function returns a new record with type *type* and *length* more slots, each initialized to *object*.

```
(setq sleepy (make-record 'foo 9 'Z))
     ⇒ #s(foo Z Z Z Z Z Z Z Z Z)
```

7.2 Backward Compatibility

Code compiled with older versions of `cl-defstruct` that doesn't use records may run into problems when used in a new Emacs. To alleviate this, Emacs detects when an old `cl-defstruct` is used, and enables a mode in which `type-of` handles old struct objects as if they were records.

`cl-old-struct-compat-mode` *arg* [Function]

If *arg* is positive, enable backward compatibility with old-style structs.

8 Hash Tables

A hash table is a very fast kind of lookup table, somewhat like an alist (see Section 5.8 [Association Lists], page 95) in that it maps keys to corresponding values. It differs from an alist in these ways:

- Lookup in a hash table is extremely fast for large tables—in fact, the time required is essentially *independent* of how many elements are stored in the table. For smaller tables (a few tens of elements) alists may still be faster because hash tables have a more-or-less constant overhead.
- The correspondences in a hash table are in no particular order.
- There is no way to share structure between two hash tables, the way two alists can share a common tail.

Emacs Lisp provides a general-purpose hash table data type, along with a series of functions for operating on them. Hash tables have a special printed representation, which consists of '`#s`' followed by a list specifying the hash table properties and contents. See Section 8.1 [Creating Hash], page 127. (Hash notation, the initial '`#`' character used in the printed representations of objects with no read representation, has nothing to do with hash tables. See Section 2.1 [Printed Representation], page 8.)

Obarrays are also a kind of hash table, but they are a different type of object and are used only for recording interned symbols (see Section 9.3 [Creating Symbols], page 135).

8.1 Creating Hash Tables

The principal function for creating a hash table is `make-hash-table`.

`make-hash-table` **&rest** *keyword-args* [Function]

This function creates a new hash table according to the specified arguments. The arguments should consist of alternating keywords (particular symbols recognized specially) and values corresponding to them.

Several keywords make sense in `make-hash-table`, but the only two that you really need to know about are `:test` and `:weakness`.

`:test` *test*

This specifies the method of key lookup for this hash table. The default is `eql`; `eq` and `equal` are other alternatives:

`eql` Keys which are numbers are the same if they are `equal`, that is, if they are equal in value and either both are integers or both are floating point; otherwise, two distinct objects are never the same.

`eq` Any two distinct Lisp objects are different as keys.

`equal` Two Lisp objects are the same, as keys, if they are equal according to `equal`.

You can use `define-hash-table-test` (see Section 8.3 [Defining Hash], page 130) to define additional possibilities for *test*.

`:weakness` *weak*

The weakness of a hash table specifies whether the presence of a key or value in the hash table preserves it from garbage collection.

The value, *weak*, must be one of `nil`, `key`, `value`, `key-or-value`, `key-and-value`, or `t` which is an alias for `key-and-value`. If *weak* is `key` then the hash table does not prevent its keys from being collected as garbage (if they are not referenced anywhere else); if a particular key does get collected, the corresponding association is removed from the hash table.

If *weak* is `value`, then the hash table does not prevent values from being collected as garbage (if they are not referenced anywhere else); if a particular value does get collected, the corresponding association is removed from the hash table.

If *weak* is `key-and-value` or `t`, both the key and the value must be live in order to preserve the association. Thus, the hash table does not protect either keys or values from garbage collection; if either one is collected as garbage, that removes the association.

If *weak* is `key-or-value`, either the key or the value can preserve the association. Thus, associations are removed from the hash table when both their key and value would be collected as garbage (if not for references from weak hash tables).

The default for *weak* is `nil`, so that all keys and values referenced in the hash table are preserved from garbage collection.

`:size` *size*

This specifies a hint for how many associations you plan to store in the hash table. If you know the approximate number, you can make things a little more efficient by specifying it this way. If you specify too small a size, the hash table will grow automatically when necessary, but doing that takes some extra time.

The default size is 65.

`:rehash-size` *rehash-size*

When you add an association to a hash table and the table is full, it grows automatically. This value specifies how to make the hash table larger, at that time.

If *rehash-size* is an integer, it should be positive, and the hash table grows by adding approximately that much to the nominal size. If *rehash-size* is floating point, it had better be greater than 1, and the hash table grows by multiplying the old size by approximately that number.

The default value is 1.5.

`:rehash-threshold` *threshold*

This specifies the criterion for when the hash table is full (so it should be made larger). The value, *threshold*, should be a positive floating-point number, no greater than 1. The hash table is full whenever the actual

number of entries exceeds the nominal size multiplied by an approximation to this value. The default for *threshold* is 0.8125.

You can also create a hash table using the printed representation for hash tables. The Lisp reader can read this printed representation, provided each element in the specified hash table has a valid read syntax (see Section 2.1 [Printed Representation], page 8). For instance, the following specifies a hash table containing the keys `key1` and `key2` (both symbols) associated with `val1` (a symbol) and `300` (a number) respectively.

```
#s(hash-table size 30 data (key1 val1 key2 300))
```

Note, however, that when using this in Emacs Lisp code, it's undefined whether this creates a new hash table or not. If you want to create a new hash table, you should always use `make-hash-table` (see Section 10.1.1 [Self-Evaluating Forms], page 146).

The printed representation for a hash table consists of '`#s`' followed by a list beginning with '`hash-table`'. The rest of the list should consist of zero or more property-value pairs specifying the hash table's properties and initial contents. The properties and values are read literally. Valid property names are `size`, `test`, `weakness`, `rehash-size`, `rehash-threshold`, and `data`. The `data` property should be a list of key-value pairs for the initial contents; the other properties have the same meanings as the matching `make-hash-table` keywords (`:size`, `:test`, etc.), described above.

Note that you cannot specify a hash table whose initial contents include objects that have no read syntax, such as buffers and frames. Such objects may be added to the hash table after it is created.

8.2 Hash Table Access

This section describes the functions for accessing and storing associations in a hash table. In general, any Lisp object can be used as a hash key, unless the comparison method imposes limits. Any Lisp object can also be used as the value.

`gethash` *key table* **&optional** *default* [Function]
: This function looks up *key* in *table*, and returns its associated *value*—or *default*, if *key* has no association in *table*.

`puthash` *key value table* [Function]
: This function enters an association for *key* in *table*, with value *value*. If *key* already has an association in *table*, *value* replaces the old associated value. This function always returns *value*.

`remhash` *key table* [Function]
: This function removes the association for *key* from *table*, if there is one. If *key* has no association, `remhash` does nothing.

  Common Lisp note: In Common Lisp, `remhash` returns non-`nil` if it actually removed an association and `nil` otherwise. In Emacs Lisp, `remhash` always returns `nil`.

`clrhash` *table* [Function]
: This function removes all the associations from hash table *table*, so that it becomes empty. This is also called *clearing* the hash table. `clrhash` returns the empty *table*.

`maphash` *function table* [Function]

This function calls *function* once for each of the associations in *table*. The function *function* should accept two arguments—a *key* listed in *table*, and its associated *value*. `maphash` returns `nil`.

8.3 Defining Hash Comparisons

You can define new methods of key lookup by means of `define-hash-table-test`. In order to use this feature, you need to understand how hash tables work, and what a *hash code* means.

You can think of a hash table conceptually as a large array of many slots, each capable of holding one association. To look up a key, `gethash` first computes an integer, the hash code, from the key. It can reduce this integer modulo the length of the array, to produce an index in the array. Then it looks in that slot, and if necessary in other nearby slots, to see if it has found the key being sought.

Thus, to define a new method of key lookup, you need to specify both a function to compute the hash code from a key, and a function to compare two keys directly. The two functions should be consistent with each other: that is, two keys' hash codes should be the same if the keys compare as equal. Also, since the two functions can be called at any time (such as by the garbage collector), the functions should be free of side effects and should return quickly, and their behavior should depend on only on properties of the keys that do not change.

`define-hash-table-test` *name test-fn hash-fn* [Function]

This function defines a new hash table test, named *name*.

After defining *name* in this way, you can use it as the *test* argument in `make-hash-table`. When you do that, the hash table will use *test-fn* to compare key values, and *hash-fn* to compute a hash code from a key value.

The function *test-fn* should accept two arguments, two keys, and return non-`nil` if they are considered the same.

The function *hash-fn* should accept one argument, a key, and return an integer that is the hash code of that key. For good results, the function should use the whole range of fixnums for hash codes, including negative fixnums.

The specified functions are stored in the property list of *name* under the property `hash-table-test`; the property value's form is `(`*`test-fn hash-fn`*`)`.

`sxhash-equal` *obj* [Function]

This function returns a hash code for Lisp object *obj*. This is an integer that reflects the contents of *obj* and the other Lisp objects it points to.

If two objects *obj1* and *obj2* are `equal`, then `(sxhash-equal` *obj1*`)` and `(sxhash-equal` *obj2*`)` are the same integer.

If the two objects are not `equal`, the values returned by `sxhash-equal` are usually different, but not always. `sxhash-equal` is designed to be reasonably fast (since it's used for indexing hash tables) so it won't recurse deeply into nested structures. In addition; once in a rare while, by luck, you will encounter two distinct-looking simple

objects that give the same result from `sxhash-equal`. So you can't, in general, use `sxhash-equal` to check whether an object has changed.

Common Lisp note: In Common Lisp a similar function is called `sxhash`. Emacs provides this name as a compatibility alias for `sxhash-equal`.

`sxhash-eq` *obj* [Function]

This function returns a hash code for Lisp object *obj*. Its result reflects identity of *obj*, but not its contents.

If two objects *obj1* and *obj2* are `eq`, then `(sxhash-eq obj1)` and `(sxhash-eq obj2)` are the same integer.

`sxhash-eql` *obj* [Function]

This function returns a hash code for Lisp object *obj* suitable for `eql` comparison. I.e. it reflects identity of *obj* except for the case where the object is a bignum or a float number, in which case a hash code is generated for the value.

If two objects *obj1* and *obj2* are `eql`, then `(sxhash-eql obj1)` and `(sxhash-eql obj2)` are the same integer.

This example creates a hash table whose keys are strings that are compared case-insensitively.

```
(defun string-hash-ignore-case (a)
  (sxhash-equal (upcase a)))

(define-hash-table-test 'ignore-case
  'string-equal-ignore-case 'string-hash-ignore-case)

(make-hash-table :test 'ignore-case)
```

Here is how you could define a hash table test equivalent to the predefined test value `equal`. The keys can be any Lisp object, and equal-looking objects are considered the same key.

```
(define-hash-table-test 'contents-hash 'equal 'sxhash-equal)

(make-hash-table :test 'contents-hash)
```

Lisp programs should *not* rely on hash codes being preserved between Emacs sessions, as the implementation of the hash functions uses some details of the object storage that can change between sessions and between different architectures.

8.4 Other Hash Table Functions

Here are some other functions for working with hash tables.

`hash-table-p` *table* [Function]

This returns non-`nil` if *table* is a hash table object.

`copy-hash-table` *table* [Function]

This function creates and returns a copy of *table*. Only the table itself is copied—the keys and values are shared.

`hash-table-count` *table* [Function]
This function returns the actual number of entries in *table*.

`hash-table-test` *table* [Function]
This returns the *test* value that was given when *table* was created, to specify how to hash and compare keys. See `make-hash-table` (see Section 8.1 [Creating Hash], page 127).

`hash-table-weakness` *table* [Function]
This function returns the *weak* value that was specified for hash table *table*.

`hash-table-rehash-size` *table* [Function]
This returns the rehash size of *table*.

`hash-table-rehash-threshold` *table* [Function]
This returns the rehash threshold of *table*.

`hash-table-size` *table* [Function]
This returns the current nominal size of *table*.

9 Symbols

A *symbol* is an object with a unique name. This chapter describes symbols, their components, their property lists, and how they are created and interned. Separate chapters describe the use of symbols as variables and as function names; see Chapter 12 [Variables], page 189, and Chapter 13 [Functions], page 230. For the precise read syntax for symbols, see Section 2.4.4 [Symbol Type], page 14.

You can test whether an arbitrary Lisp object is a symbol with `symbolp`:

`symbolp` *object* [Function]
: This function returns `t` if *object* is a symbol, `nil` otherwise.

9.1 Symbol Components

Each symbol has four components (or "cells"), each of which references another object:

Print name
: The symbol's name.

Value
: The symbol's current value as a variable.

Function
: The symbol's function definition. It can also hold a symbol, a keymap, or a keyboard macro.

Property list
: The symbol's property list.

The print name cell always holds a string, and cannot be changed. Each of the other three cells can be set to any Lisp object.

The print name cell holds the string that is the name of a symbol. Since symbols are represented textually by their names, it is important not to have two symbols with the same name. The Lisp reader ensures this: every time it reads a symbol, it looks for an existing symbol with the specified name before it creates a new one. To get a symbol's name, use the function `symbol-name` (see Section 9.3 [Creating Symbols], page 135). However, although each symbol has only one unique *print name*, it is nevertheless possible to refer to that same symbol via different alias names called "shorthands" (see Section 9.5 [Shorthands], page 142).

The value cell holds a symbol's value as a variable, which is what you get if the symbol itself is evaluated as a Lisp expression. See Chapter 12 [Variables], page 189, for details about how values are set and retrieved, including complications such as *local bindings* and *scoping rules*. Most symbols can have any Lisp object as a value, but certain special symbols have values that cannot be changed; these include `nil` and `t`, and any symbol whose name starts with '`:`' (those are called *keywords*). See Section 12.2 [Constant Variables], page 189.

The function cell holds a symbol's function definition. Often, we refer to "the function `foo`" when we really mean the function stored in the function cell of `foo`; we make the distinction explicit only when necessary. Typically, the function cell is used to hold a function (see Chapter 13 [Functions], page 230) or a macro (see Chapter 14 [Macros], page 269). However, it can also be used to hold a symbol (see Section 10.1.4 [Function Indirection], page 147), keyboard macro (see Section 22.16 [Keyboard Macros], page 479),

keymap (see Chapter 23 [Keymaps], page 481), or autoload object (see Section 10.1.8 [Autoloading], page 150). To get the contents of a symbol's function cell, use the function `symbol-function` (see Section 13.9 [Function Cells], page 248).

The property list cell normally should hold a correctly formatted property list. To get a symbol's property list, use the function `symbol-plist`. See Section 9.4 [Symbol Properties], page 138.

The function cell or the value cell may be *void*, which means that the cell does not reference any object. (This is not the same thing as holding the symbol `void`, nor the same as holding the symbol `nil`.) Examining a function or value cell that is void results in an error, such as '`Symbol's value as variable is void`'.

Because each symbol has separate value and function cells, variables names and function names do not conflict. For example, the symbol `buffer-file-name` has a value (the name of the file being visited in the current buffer) as well as a function definition (a primitive function that returns the name of the file):

```
buffer-file-name
     ⇒ "/gnu/elisp/symbols.texi"
(symbol-function 'buffer-file-name)
     ⇒ #<subr buffer-file-name>
```

9.2 Defining Symbols

A *definition* is a special kind of Lisp expression that announces your intention to use a symbol in a particular way. It typically specifies a value or meaning for the symbol for one kind of use, plus documentation for its meaning when used in this way. Thus, when you define a symbol as a variable, you can supply an initial value for the variable, plus documentation for the variable.

`defvar` and `defconst` are special forms that define a symbol as a *global variable*—a variable that can be accessed at any point in a Lisp program. See Chapter 12 [Variables], page 189, for details about variables. To define a customizable variable, use the `defcustom` macro, which also calls `defvar` as a subroutine (see Chapter 15 [Customization], page 278).

In principle, you can assign a variable value to any symbol with `setq`, whether or not it has first been defined as a variable. However, you ought to write a variable definition for each global variable that you want to use; otherwise, your Lisp program may not act correctly if it is evaluated with lexical scoping enabled (see Section 12.10 [Variable Scoping], page 201).

`defun` defines a symbol as a function, creating a lambda expression and storing it in the function cell of the symbol. This lambda expression thus becomes the function definition of the symbol. (The term "function definition", meaning the contents of the function cell, is derived from the idea that `defun` gives the symbol its definition as a function.) `defsubst` and `defalias` are two other ways of defining a function. See Chapter 13 [Functions], page 230.

`defmacro` defines a symbol as a macro. It creates a macro object and stores it in the function cell of the symbol. Note that a given symbol can be a macro or a function, but not both at once, because both macro and function definitions are kept in the function cell, and that cell can hold only one Lisp object at any given time. See Chapter 14 [Macros], page 269.

As previously noted, Emacs Lisp allows the same symbol to be defined both as a variable (e.g., with `defvar`) and as a function or macro (e.g., with `defun`). Such definitions do not conflict.

These definitions also act as guides for programming tools. For example, the `C-h f` and `C-h v` commands create help buffers containing links to the relevant variable, function, or macro definitions. See Section "Name Help" in *The GNU Emacs Manual*.

9.3 Creating and Interning Symbols

To understand how symbols are created in GNU Emacs Lisp, you must know how Lisp reads them. Lisp must ensure that it finds the same symbol every time it reads the same sequence of characters in the same context. Failure to do so would cause complete confusion.

When the Lisp reader encounters a name that references a symbol in the source code, it reads all the characters of that name. Then it looks up that name in a table called an *obarray* to find the symbol that the programmer meant. The technique used in this lookup is called "hashing", an efficient method of looking something up by converting a sequence of characters to a number, known as a "hash code". For example, instead of searching a telephone book cover to cover when looking up Jan Jones, you start with the J's and go from there. That is a simple version of hashing. Each element of the obarray is a *bucket* which holds all the symbols with a given hash code; to look for a given name, it is sufficient to look through all the symbols in the bucket for that name's hash code. (The same idea is used for general Emacs hash tables, but they are a different data type; see Chapter 8 [Hash Tables], page 127.)

When looking up names, the Lisp reader also considers "shorthands". If the programmer supplied them, this allows the reader to find a symbol even if its name isn't present in its full form in the source code. Of course, the reader needs to be aware of some pre-established context about such shorthands, much as one needs context to be to able to refer uniquely to Jan Jones by just the name "Jan": it's probably fine when amongst the Joneses, or when Jan has been mentioned recently, but very ambiguous in any other situation. See Section 9.5 [Shorthands], page 142.

If a symbol with the desired name is found, the reader uses that symbol. If the obarray does not contain a symbol with that name, the reader makes a new symbol and adds it to the obarray. Finding or adding a symbol with a certain name is called *interning* it, and the symbol is then called an *interned symbol*.

Interning ensures that each obarray has just one symbol with any particular name. Other like-named symbols may exist, but not in the same obarray. Thus, the reader gets the same symbols for the same names, as long as you keep reading with the same obarray.

Interning usually happens automatically in the reader, but sometimes other programs may want to do it. For example, after the `M-x` command obtains the command name as a string using the minibuffer, it then interns the string, to get the interned symbol with that name. As another example, a hypothetical telephone book program could intern the name of each looked up person's name as a symbol, even if the obarray did not contain it, so that it could attach information to that new symbol, such as the last time someone looked it up.

No obarray contains all symbols; in fact, some symbols are not in any obarray. They are called *uninterned symbols*. An uninterned symbol has the same four cells as other symbols; however, the only way to gain access to it is by finding it in some other object or as the

value of a variable. Uninterned symbols are sometimes useful in generating Lisp code, see below.

In Emacs Lisp, an obarray is actually a vector. Each element of the vector is a bucket; its value is either an interned symbol whose name hashes to that bucket, or 0 if the bucket is empty. Each interned symbol has an internal link (invisible to the user) to the next symbol in the bucket. Because these links are invisible, there is no way to find all the symbols in an obarray except using `mapatoms` (below). The order of symbols in a bucket is not significant.

In an empty obarray, every element is 0, so you can create an obarray with `(make-vector` *`length`* `0)`. **This is the only valid way to create an obarray.** Prime numbers as lengths tend to result in good hashing; lengths one less than a power of two are also good.

Do not try to put symbols in an obarray yourself. This does not work—only `intern` can enter a symbol in an obarray properly.

> **Common Lisp note:** Unlike Common Lisp, Emacs Lisp does not provide for interning the same name in several different "packages", thus creating multiple symbols with the same name but different packages. Emacs Lisp provides a different namespacing system called "shorthands" (see Section 9.5 [Shorthands], page 142).

Most of the functions below take a name and sometimes an obarray as arguments. A `wrong-type-argument` error is signaled if the name is not a string, or if the obarray is not a vector.

`symbol-name` *symbol* [Function]

This function returns the string that is *symbol*'s name. For example:

```
(symbol-name 'foo)
     ⇒ "foo"
```

Warning: Never alter the string returned by that function. Doing that might make Emacs dysfunctional, and might even crash Emacs.

Creating an uninterned symbol is useful in generating Lisp code, because an uninterned symbol used as a variable in the code you generate cannot clash with any variables used in other Lisp programs.

`make-symbol` *name* [Function]

This function returns a newly-allocated, uninterned symbol whose name is *name* (which must be a string). Its value and function definition are void, and its property list is `nil`. In the example below, the value of `sym` is not `eq` to `foo` because it is a distinct uninterned symbol whose name is also '`foo`'.

```
(setq sym (make-symbol "foo"))
     ⇒ foo
(eq sym 'foo)
     ⇒ nil
```

`gensym` **&optional** *prefix* [Function]

This function returns a symbol using `make-symbol`, whose name is made by appending `gensym-counter` to *prefix* and incrementing that counter, guaranteeing that no two calls to this function will generate a symbol with the same name. The prefix defaults to `"g"`.

To avoid problems when accidentally interning printed representation of generated code (see Section 2.1 [Printed Representation], page 8), it is recommended to use `gensym` instead of `make-symbol`.

`intern` *name* **&optional** *obarray* [Function]

This function returns the interned symbol whose name is *name*. If there is no such symbol in the obarray *obarray*, `intern` creates a new one, adds it to the obarray, and returns it. If *obarray* is omitted, the value of the global variable `obarray` is used.

```
(setq sym (intern "foo"))
     ⇒ foo
(eq sym 'foo)
     ⇒ t

(setq sym1 (intern "foo" other-obarray))
     ⇒ foo
(eq sym1 'foo)
     ⇒ nil
```

Common Lisp note: In Common Lisp, you can intern an existing symbol in an obarray. In Emacs Lisp, you cannot do this, because the argument to `intern` must be a string, not a symbol.

`intern-soft` *name* **&optional** *obarray* [Function]

This function returns the symbol in *obarray* whose name is *name*, or `nil` if *obarray* has no symbol with that name. Therefore, you can use `intern-soft` to test whether a symbol with a given name is already interned. If *obarray* is omitted, the value of the global variable `obarray` is used.

The argument *name* may also be a symbol; in that case, the function returns *name* if *name* is interned in the specified obarray, and otherwise `nil`.

```
(intern-soft "frazzle")         ; No such symbol exists.
     ⇒ nil
(make-symbol "frazzle")         ; Create an uninterned one.
     ⇒ frazzle
(intern-soft "frazzle")         ; That one cannot be found.
     ⇒ nil
(setq sym (intern "frazzle"))   ; Create an interned one.
     ⇒ frazzle
(intern-soft "frazzle")         ; That one can be found!
     ⇒ frazzle
(eq sym 'frazzle)               ; And it is the same one.
     ⇒ t
```

`obarray` [Variable]

This variable is the standard obarray for use by `intern` and `read`.

`mapatoms` *function* **&optional** *obarray* [Function]

This function calls *function* once with each symbol in the obarray *obarray*. Then it returns `nil`. If *obarray* is omitted, it defaults to the value of `obarray`, the standard obarray for ordinary symbols.

```
(setq count 0)
     ⇒ 0
(defun count-syms (s)
  (setq count (1+ count)))
     ⇒ count-syms
(mapatoms 'count-syms)
     ⇒ nil
count
     ⇒ 1871
```

See `documentation` in Section 25.2 [Accessing Documentation], page 598, for another example using `mapatoms`.

`unintern` *symbol obarray* [Function]

This function deletes *symbol* from the obarray *obarray*. If `symbol` is not actually in the obarray, `unintern` does nothing. If *obarray* is `nil`, the current obarray is used.

If you provide a string instead of a symbol as *symbol*, it stands for a symbol name. Then `unintern` deletes the symbol (if any) in the obarray which has that name. If there is no such symbol, `unintern` does nothing.

If `unintern` does delete a symbol, it returns `t`. Otherwise it returns `nil`.

9.4 Symbol Properties

A symbol may possess any number of *symbol properties*, which can be used to record miscellaneous information about the symbol. For example, when a symbol has a `risky-local-variable` property with a non-`nil` value, that means the variable which the symbol names is a risky file-local variable (see Section 12.12 [File Local Variables], page 213).

Each symbol's properties and property values are stored in the symbol's property list cell (see Section 9.1 [Symbol Components], page 133), in the form of a property list (see Section 5.9 [Property Lists], page 99).

9.4.1 Accessing Symbol Properties

The following functions can be used to access symbol properties.

`get` *symbol property* [Function]

This function returns the value of the property named *property* in *symbol*'s property list. If there is no such property, it returns `nil`. Thus, there is no distinction between a value of `nil` and the absence of the property.

The name *property* is compared with the existing property names using `eq`, so any object is a legitimate property.

See `put` for an example.

`put` *symbol property value* [Function]

This function puts *value* onto *symbol*'s property list under the property name *property*, replacing any previous property value. The `put` function returns *value*.

```
(put 'fly 'verb 'transitive)
     ⇒'transitive
```

```
(put 'fly 'noun '(a buzzing little bug))
     ⇒ (a buzzing little bug)
(get 'fly 'verb)
     ⇒ transitive
(symbol-plist 'fly)
     ⇒ (verb transitive noun (a buzzing little bug))
```

`symbol-plist` *symbol* [Function]
This function returns the property list of *symbol*.

`setplist` *symbol plist* [Function]
This function sets *symbol*'s property list to *plist*. Normally, *plist* should be a well-formed property list, but this is not enforced. The return value is *plist*.

```
(setplist 'foo '(a 1 b (2 3) c nil))
     ⇒ (a 1 b (2 3) c nil)
(symbol-plist 'foo)
     ⇒ (a 1 b (2 3) c nil)
```

For symbols in special obarrays, which are not used for ordinary purposes, it may make sense to use the property list cell in a nonstandard fashion; in fact, the abbrev mechanism does so (see Chapter 38 [Abbrevs], page 1069).

You could define `put` in terms of `setplist` and `plist-put`, as follows:

```
(defun put (symbol prop value)
  (setplist symbol
            (plist-put (symbol-plist symbol) prop value)))
```

`function-get` *symbol property* **&optional** *autoload* [Function]
This function is identical to `get`, except that if *symbol* is the name of a function alias, it looks in the property list of the symbol naming the actual function. See Section 13.4 [Defining Functions], page 237. If the optional argument *autoload* is non-`nil`, and *symbol* is auto-loaded, this function will try to autoload it, since autoloading might set *property* of *symbol*. If *autoload* is the symbol `macro`, only try autoloading if *symbol* is an auto-loaded macro.

`function-put` *function property value* [Function]
This function sets *property* of *function* to *value*. *function* should be a symbol. This function is preferred to calling `put` for setting properties of a function, because it will allow us some day to implement remapping of old properties to new ones.

9.4.2 Standard Symbol Properties

Here, we list the symbol properties which are used for special purposes in Emacs. In the following table, whenever we say "the named function", that means the function whose name is the relevant symbol; similarly for "the named variable" etc.

`:advertised-binding`
This property value specifies the preferred key binding, when showing documentation, for the named function. See Section 25.3 [Keys in Documentation], page 600.

`char-table-extra-slots`
: The value, if non-`nil`, specifies the number of extra slots in the named char-table type. See Section 6.6 [Char-Tables], page 119.

`customized-face`
`face-defface-spec`
`saved-face`
`theme-face`
: These properties are used to record a face's standard, saved, customized, and themed face specs. Do not set them directly; they are managed by `defface` and related functions. See Section 41.12.2 [Defining Faces], page 1172.

`customized-value`
`saved-value`
`standard-value`
`theme-value`
: These properties are used to record a customizable variable's standard value, saved value, customized-but-unsaved value, and themed values. Do not set them directly; they are managed by `defcustom` and related functions. See Section 15.3 [Variable Definitions], page 281.

`definition-name`
: This property is used to find the definition of a symbol in the source code, when it might be hard to find the definition by textual search of the source file. For example, a `define-derived-mode` (see Section 24.2.4 [Derived Modes], page 535) might define a mode-specific function or a variable implicitly; or your Lisp program might generate a run-time call to `defun` to define a function (see Section 13.4 [Defining Functions], page 237). In these and similar cases, the `definition-name` property of the symbol should be another symbol whose definition can be found by textual search and whose code defines the original symbol. In the example with `define-derived-mode`, the value of this property of the functions and variables it defines should be the mode symbol. The Emacs Help commands such as `C-h f` (see Section "Help" in *The GNU Emacs Manual*) use this property to show the definition of a symbol via a button in the `*Help*` buffer where the symbol's documentation is shown.

`disabled` If the value is non-`nil`, the named function is disabled as a command. See Section 22.14 [Disabling Commands], page 478.

`face-documentation`
: The value stores the documentation string of the named face. This is set automatically by `defface`. See Section 41.12.2 [Defining Faces], page 1172.

`history-length`
: The value, if non-`nil`, specifies the maximum minibuffer history length for the named history list variable. See Section 21.4 [Minibuffer History], page 393.

`interactive-form`
: The value is an interactive form for the named function. Normally, you should not set this directly; use the `interactive` special form instead. See Section 22.3 [Interactive Call], page 433.

`menu-enable`
: The value is an expression for determining whether the named menu item should be enabled in menus. See Section 23.18.1.1 [Simple Menu Items], page 511.

`mode-class`
: If the value is `special`, the named major mode is special. See Section 24.2.1 [Major Mode Conventions], page 529.

`permanent-local`
: If the value is non-`nil`, the named variable is a buffer-local variable whose value should not be reset when changing major modes. See Section 12.11.2 [Creating Buffer-Local], page 208.

`permanent-local-hook`
: If the value is non-`nil`, the named function should not be deleted from the local value of a hook variable when changing major modes. See Section 24.1.2 [Setting Hooks], page 526.

`pure`
: If the value is non-`nil`, the named function is considered to be pure (see Section 13.1 [What Is a Function], page 230). Calls with constant arguments can be evaluated at compile time. This may shift run time errors to compile time. Not to be confused with pure storage (see Section E.2 [Pure Storage], page 1354).

`risky-local-variable`
: If the value is non-`nil`, the named variable is considered risky as a file-local variable. See Section 12.12 [File Local Variables], page 213.

`safe-function`
: If the value is non-`nil`, the named function is considered generally safe for evaluation. See Section 13.17 [Function Safety], page 266.

`safe-local-eval-function`
: If the value is non-`nil`, the named function is safe to call in file-local evaluation forms. See Section 12.12 [File Local Variables], page 213.

`safe-local-variable`
: The value specifies a function for determining safe file-local values for the named variable. See Section 12.12 [File Local Variables], page 213. Since this value is consulted when loading files, the function should be efficient and should ideally not lead to loading any libraries to determine the safeness (e.g., it should not be an autoloaded function).

`side-effect-free`
: A non-`nil` value indicates that the named function is free of side effects (see Section 13.1 [What Is a Function], page 230), so the byte compiler may ignore a call whose value is unused. If the property's value is `error-free`, the byte compiler may even delete such unused calls. In addition to byte compiler optimizations, this property is also used for determining function safety (see Section 13.17 [Function Safety], page 266).

`undo-inhibit-region`
: If non-`nil`, the named function prevents the `undo` operation from being restricted to the active region, if `undo` is invoked immediately after the function. See Section 33.9 [Undo], page 898.

`variable-documentation`
: If non-`nil`, this specifies the named variable's documentation string. This is set automatically by `defvar` and related functions. See Section 41.12.2 [Defining Faces], page 1172.

9.5 Shorthands

The symbol *shorthands*, sometimes known as "renamed symbols", are symbolic forms found in Lisp source. They're just like regular symbolic forms, except that when the Lisp reader encounters them, it produces symbols which have a different and usually longer *print name* (see Section 9.1 [Symbol Components], page 133).

It is useful to think of shorthands as *abbreviating* the full names of intended symbols. Despite this, do not confuse shorthands with the Abbrev system (see Chapter 38 [Abbrevs], page 1069).

Shorthands make Emacs Lisp's *namespacing etiquette* easier to work with. Since all symbols are stored in a single obarray (see Section 9.3 [Creating Symbols], page 135), programmers commonly prefix each symbol name with the name of the library where it originates. For example, the functions `text-property-search-forward` and `text-property-search-backward` both belong to the `text-property-search.el` library (see Chapter 16 [Loading], page 299). By properly prefixing symbol names, one effectively prevents clashes between similarly named symbols which belong to different libraries and thus do different things. However, this practice commonly originates very long symbols names, which are inconvenient to type and read after a while. Shorthands solve these issues in a clean way.

`read-symbol-shorthands` [Variable]
: This variable's value is an alist whose elements have the form `(`*`shorthand-prefix`* `.` *`longhand-prefix`*`)`. Each element instructs the Lisp reader to read every symbol form which starts with *shorthand-prefix* as if it started with *longhand-prefix* instead.

  This variable may only be set in file-local variables (see Section "Local Variables in Files" in *The GNU Emacs Manual*).

Here's an example of shorthands usage in a hypothetical string manipulating library `some-nice-string-utils.el`.

```
(defun some-nice-string-utils-split (separator s &optional omit-nulls)
  "A match-data saving variant of `split-string'."
  (save-match-data (split-string s separator omit-nulls)))

(defun some-nice-string-utils-lines (s)
  "Split string S at newline characters into a list of strings."
  (some-nice-string-utils-split "\\(\r\n\\|[\n\r]\\)" s))
```

As can be seen, it's quite tedious to read or develop this code since the symbol names to type are so long. We can use shorthands to alleviate that.

```
(defun snu-split (separator s &optional omit-nulls)
```

```
  "A match-data saving variation on `split-string'."
  (save-match-data (split-string s separator omit-nulls)))

(defun snu-lines (s)
  "Split string S into a list of strings on newline characters."
  (snu-split "\\(\r\n\\|[\n\r]\\)" s))

;; Local Variables:
;; read-symbol-shorthands: (("snu-" . "some-nice-string-utils-"))
;; End:
```

Even though the two excerpts look different, they are quite identical after the Lisp reader processes them. Both will lead to the very same symbols being interned (see Section 9.3 [Creating Symbols], page 135). Thus loading or byte-compiling any of the two files has equivalent results. The shorthands `snu-split` and `snu-lines` used in the second version are *not* interned in the obarray. This is easily seen by moving point to the location where the shorthands are used and waiting for ElDoc (see Section "Local Variables in Files" in *The GNU Emacs Manual*) to hint at the true full name of the symbol under point in the echo area.

Since `read-symbol-shorthands` is a file-local variable, it is possible that multiple libraries depending on `some-nice-string-utils-lines.el` refer to the same symbols under *different* shorthands, or not using shorthands at all. In the next example, the `my-tricks.el` library refers to the symbol `some-nice-string-utils-lines` using the `sns-` prefix instead of `snu-`.

```
(defun t-reverse-lines (s) (string-join (reverse (sns-lines s)) "\n")

;; Local Variables:
;; read-symbol-shorthands: (("t-" . "my-tricks-")
;;                          ("sns-" . "some-nice-string-utils-"))
;; End:
```

9.5.1 Exceptions

There are two exceptions to rules governing Shorthand transformations:

- Symbol forms comprised entirely of characters in the Emacs Lisp symbol constituent class (see Section 36.2.1 [Syntax Class Table], page 1026) are not transformed. For example, it's possible to use `-` or `/=` as shorthand prefixes, but that won't shadow the arithmetic *functions* of those names.
- Symbol forms whose names start with '`#_`' are not transformed.

9.6 Symbols with Position

A *symbol with position* is a symbol, the *bare symbol*, together with an unsigned integer called the *position*. These objects are intended for use by the byte compiler, which records in them the position of each symbol occurrence and uses those positions in warning and error messages.

The printed representation of a symbol with position uses the hash notation outlined in Section 2.1 [Printed Representation], page 8. It looks like '`#<symbol foo at 12345>`'. It has

no read syntax. You can cause just the bare symbol to be printed by binding the variable `print-symbols-bare` to non-`nil` around the print operation. The byte compiler does this before writing its output to the compiled Lisp file.

For most purposes, when the flag variable `symbols-with-pos-enabled` is non-`nil`, symbols with positions behave just as bare symbols do. For example, '`(eq #<symbol foo at 12345> foo)`' has a value `t` when that variable is set (but `nil` when it isn't set). Most of the time in Emacs this variable is `nil`, but the byte compiler binds it to `t` when it runs.

Typically, symbols with position are created by the byte compiler calling the reader function `read-positioning-symbols` (see Section 20.3 [Input Functions], page 375). One can also be created with the function `position-symbol`.

`symbols-with-pos-enabled` [Variable]
: When this variable is non-`nil`, symbols with position behave like the contained bare symbol. Emacs runs a little more slowly in this case.

`print-symbols-bare` [Variable]
: When bound to non-`nil`, the Lisp printer prints only the bare symbol of a symbol with position, ignoring the position.

`symbol-with-pos-p` *symbol.* [Function]
: This function returns `t` if *symbol* is a symbol with position, `nil` otherwise.

`bare-symbol` *symbol* [Function]
: This function returns the bare symbol contained in *symbol*, or *symbol* itself if it is already a bare symbol. For any other type of object, it signals an error.

`symbol-with-pos-pos` *symbol* [Function]
: This function returns the position, a number, from a symbol with position. For any other type of object, it signals an error.

`position-symbol` *sym pos* [Function]
: Make a new symbol with position. *sym* is either a bare symbol or a symbol with position, and supplies the symbol part of the new object. *pos* is either an integer which becomes the number part of the new object, or a symbol with position whose position is used. Emacs signals an error if either argument is invalid.

10 Evaluation

The *evaluation* of expressions in Emacs Lisp is performed by the *Lisp interpreter*—a program that receives a Lisp object as input and computes its *value as an expression*. How it does this depends on the data type of the object, according to rules described in this chapter. The interpreter runs automatically to evaluate portions of your program, but can also be called explicitly via the Lisp primitive function `eval`.

A Lisp object that is intended for evaluation is called a *form* or *expression*[1]. The fact that forms are data objects and not merely text is one of the fundamental differences between Lisp-like languages and typical programming languages. Any object can be evaluated, but in practice only numbers, symbols, lists and strings are evaluated very often.

In subsequent sections, we will describe the details of what evaluation means for each kind of form.

It is very common to read a Lisp form and then evaluate the form, but reading and evaluation are separate activities, and either can be performed alone. Reading per se does not evaluate anything; it converts the printed representation of a Lisp object to the object itself. It is up to the caller of `read` to specify whether this object is a form to be evaluated, or serves some entirely different purpose. See Section 20.3 [Input Functions], page 375.

Evaluation is a recursive process, and evaluating a form often involves evaluating parts within that form. For instance, when you evaluate a *function call* form such as `(car x)`, Emacs first evaluates the argument (the subform `x`). After evaluating the argument, Emacs *executes* the function (`car`), and if the function is written in Lisp, execution works by evaluating the *body* of the function (in this example, however, `car` is not a Lisp function; it is a primitive function implemented in C). See Chapter 13 [Functions], page 230, for more information about functions and function calls.

Evaluation takes place in a context called the *environment*, which consists of the current values and bindings of all Lisp variables (see Chapter 12 [Variables], page 189).[2] Whenever a form refers to a variable without creating a new binding for it, the variable evaluates to the value given by the current environment. Evaluating a form may also temporarily alter the environment by binding variables (see Section 12.3 [Local Variables], page 190).

Evaluating a form may also make changes that persist; these changes are called *side effects*. An example of a form that produces a side effect is `(setq foo 1)`.

Do not confuse evaluation with command key interpretation. The editor command loop translates keyboard input into a command (an interactively callable function) using the active keymaps, and then uses `call-interactively` to execute that command. Executing the command usually involves evaluation, if the command is written in Lisp; however, this step is not considered a part of command key interpretation. See Chapter 22 [Command Loop], page 424.

---

[1] It is sometimes also referred to as an *S-expression* or *sexp*, but we generally do not use this terminology in this manual.

[2] This definition of "environment" is specifically not intended to include all the data that can affect the result of a program.

10.1 Kinds of Forms

A Lisp object that is intended to be evaluated is called a *form* (or an *expression*). How Emacs evaluates a form depends on its data type. Emacs has three different kinds of form that are evaluated differently: symbols, lists, and all other types. This section describes all three kinds, one by one, starting with the other types, which are self-evaluating forms.

10.1.1 Self-Evaluating Forms

A *self-evaluating form* is any form that is not a list or symbol. Self-evaluating forms evaluate to themselves: the result of evaluation is the same object that was evaluated. Thus, the number 25 evaluates to 25, and the string `"foo"` evaluates to the string `"foo"`. Likewise, evaluating a vector does not cause evaluation of the elements of the vector—it returns the same vector with its contents unchanged.

```
'123               ; A number, shown without evaluation.
     ⇒ 123
123                ; Evaluated as usual—result is the same.
     ⇒ 123
(eval '123)        ; Evaluated "by hand"—result is the same.
     ⇒ 123
(eval (eval '123)) ; Evaluating twice changes nothing.
     ⇒ 123
```

A self-evaluating form yields a value that becomes part of the program, and you should not try to modify it via `setcar`, `aset` or similar operations. The Lisp interpreter might unify the constants yielded by your program's self-evaluating forms, so that these constants might share structure. See Section 2.9 [Mutability], page 37.

It is common to write numbers, characters, strings, and even vectors in Lisp code, taking advantage of the fact that they self-evaluate. However, it is quite unusual to do this for types that lack a read syntax, because there's no way to write them textually. It is possible to construct Lisp expressions containing these types by means of a Lisp program. Here is an example:

```
;; Build an expression containing a buffer object.
(setq print-exp (list 'print (current-buffer)))
     ⇒ (print #<buffer eval.texi>)
;; Evaluate it.
(eval print-exp)
     ⊣ #<buffer eval.texi>
     ⇒ #<buffer eval.texi>
```

10.1.2 Symbol Forms

When a symbol is evaluated, it is treated as a variable. The result is the variable's value, if it has one. If the symbol has no value as a variable, the Lisp interpreter signals an error. For more information on the use of variables, see Chapter 12 [Variables], page 189.

In the following example, we set the value of a symbol with `setq`. Then we evaluate the symbol, and get back the value that `setq` stored.

```
(setq a 123)
     ⇒ 123
```

```
(eval 'a)
     ⇒ 123
a
     ⇒ 123
```

The symbols `nil` and `t` are treated specially, so that the value of `nil` is always `nil`, and the value of `t` is always `t`; you cannot set or bind them to any other values. Thus, these two symbols act like self-evaluating forms, even though `eval` treats them like any other symbol. A symbol whose name starts with '`:`' also self-evaluates in the same way; likewise, its value ordinarily cannot be changed. See Section 12.2 [Constant Variables], page 189.

10.1.3 Classification of List Forms

A form that is a nonempty list is either a function call, a macro call, or a special form, according to its first element. These three kinds of forms are evaluated in different ways, described below. The remaining list elements constitute the *arguments* for the function, macro, or special form.

The first step in evaluating a nonempty list is to examine its first element. This element alone determines what kind of form the list is and how the rest of the list is to be processed. The first element is *not* evaluated, as it would be in some Lisp dialects such as Scheme.

10.1.4 Symbol Function Indirection

If the first element of the list is a symbol then evaluation examines the symbol's function cell, and uses its contents instead of the original symbol. If the contents are another symbol, this process, called *symbol function indirection*, is repeated until it obtains a non-symbol. See Section 13.3 [Function Names], page 236, for more information about symbol function indirection.

One possible consequence of this process is an infinite loop, in the event that a symbol's function cell refers to the same symbol. Otherwise, we eventually obtain a non-symbol, which ought to be a function or other suitable object.

More precisely, we should now have a Lisp function (a lambda expression), a byte-code function, a primitive function, a Lisp macro, a special form, or an autoload object. Each of these types is a case described in one of the following sections. If the object is not one of these types, Emacs signals an `invalid-function` error.

The following example illustrates the symbol indirection process. We use `fset` to set the function cell of a symbol and `symbol-function` to get the function cell contents (see Section 13.9 [Function Cells], page 248). Specifically, we store the symbol `car` into the function cell of `first`, and the symbol `first` into the function cell of `erste`.

```
;; Build this function cell linkage:
;;   -------------       -----        -------        -------
;;  | #<subr car> | <-- | car |  <-- | first |  <-- | erste |
;;   -------------       -----        -------        -------
(symbol-function 'car)
     ⇒ #<subr car>
(fset 'first 'car)
     ⇒ car
(fset 'erste 'first)
     ⇒ first
```

```
(erste '(1 2 3))    ; Call the function referenced by erste.
     ⇒ 1
```

By contrast, the following example calls a function without any symbol function indirection, because the first element is an anonymous Lisp function, not a symbol.

```
((lambda (arg) (erste arg))
 '(1 2 3))
     ⇒ 1
```

Executing the function itself evaluates its body; this does involve symbol function indirection when calling `erste`.

This form is rarely used and is now deprecated. Instead, you should write it as:

```
(funcall (lambda (arg) (erste arg))
         '(1 2 3))
```

or just

```
(let ((arg '(1 2 3))) (erste arg))
```

The built-in function `indirect-function` provides an easy way to perform symbol function indirection explicitly.

`indirect-function` *function* **&optional** *noerror* [Function]

This function returns the meaning of *function* as a function. If *function* is a symbol, then it finds *function*'s function definition and starts over with that value. If *function* is not a symbol, then it returns *function* itself.

This function returns `nil` if the final symbol is unbound. It signals a `cyclic-function-indirection` error if there is a loop in the chain of symbols.

The optional argument *noerror* is obsolete, kept for backward compatibility, and has no effect.

Here is how you could define `indirect-function` in Lisp:

```
(defun indirect-function (function)
  (if (and function
           (symbolp function))
      (indirect-function (symbol-function function))
    function))
```

10.1.5 Evaluation of Function Forms

If the first element of a list being evaluated is a Lisp function object, byte-code object or primitive function object, then that list is a *function call*. For example, here is a call to the function `+`:

```
(+ 1 x)
```

The first step in evaluating a function call is to evaluate the remaining elements of the list from left to right. The results are the actual argument values, one value for each list element. The next step is to call the function with this list of arguments, effectively using the function `apply` (see Section 13.5 [Calling Functions], page 239). If the function is written in Lisp, the arguments are used to bind the argument variables of the function (see Section 13.2 [Lambda Expressions], page 232); then the forms in the function body are evaluated in order, and the value of the last body form becomes the value of the function call.

10.1.6 Lisp Macro Evaluation

If the first element of a list being evaluated is a macro object, then the list is a *macro call*. When a macro call is evaluated, the elements of the rest of the list are *not* initially evaluated. Instead, these elements themselves are used as the arguments of the macro. The macro definition computes a replacement form, called the *expansion* of the macro, to be evaluated in place of the original form. The expansion may be any sort of form: a self-evaluating constant, a symbol, or a list. If the expansion is itself a macro call, this process of expansion repeats until some other sort of form results.

Ordinary evaluation of a macro call finishes by evaluating the expansion. However, the macro expansion is not necessarily evaluated right away, or at all, because other programs also expand macro calls, and they may or may not evaluate the expansions.

Normally, the argument expressions are not evaluated as part of computing the macro expansion, but instead appear as part of the expansion, so they are computed when the expansion is evaluated.

For example, given a macro defined as follows:

```
(defmacro cadr (x)
  (list 'car (list 'cdr x)))
```

an expression such as `(cadr (assq 'handler list))` is a macro call, and its expansion is:

```
(car (cdr (assq 'handler list)))
```

Note that the argument `(assq 'handler list)` appears in the expansion.

See Chapter 14 [Macros], page 269, for a complete description of Emacs Lisp macros.

10.1.7 Special Forms

A *special form* is a primitive specially marked so that its arguments are not all evaluated. Most special forms define control structures or perform variable bindings—things which functions cannot do.

Each special form has its own rules for which arguments are evaluated and which are used without evaluation. Whether a particular argument is evaluated may depend on the results of evaluating other arguments.

If an expression's first symbol is that of a special form, the expression should follow the rules of that special form; otherwise, Emacs's behavior is not well-defined (though it will not crash). For example, `((lambda (x) x . 3) 4)` contains a subexpression that begins with `lambda` but is not a well-formed `lambda` expression, so Emacs may signal an error, or may return 3 or 4 or `nil`, or may behave in other ways.

`special-form-p` *object* [Function]
: This predicate tests whether its argument is a special form, and returns `t` if so, `nil` otherwise.

Here is a list, in alphabetical order, of all of the special forms in Emacs Lisp with a reference to where each is described.

`and` see Section 11.3 [Combining Conditions], page 162,

`catch` see Section 11.7.1 [Catch and Throw], page 177,

`cond` see Section 11.2 [Conditionals], page 159,

`condition-case`
see Section 11.7.3.3 [Handling Errors], page 182,

`defconst` see Section 12.5 [Defining Variables], page 194,

`defvar` see Section 12.5 [Defining Variables], page 194,

`function` see Section 13.7 [Anonymous Functions], page 243,

`if` see Section 11.2 [Conditionals], page 159,

`interactive`
see Section 22.3 [Interactive Call], page 433,

`lambda` see Section 13.2 [Lambda Expressions], page 232,

`let`
`let*` see Section 12.3 [Local Variables], page 190,

`or` see Section 11.3 [Combining Conditions], page 162,

`prog1`
`prog2`
`progn` see Section 11.1 [Sequencing], page 158,

`quote` see Section 10.2 [Quoting], page 151,

`save-current-buffer`
see Section 28.2 [Current Buffer], page 673,

`save-excursion`
see Section 31.3 [Excursions], page 866,

`save-restriction`
see Section 31.4 [Narrowing], page 867,

`setq` see Section 12.8 [Setting Variables], page 198,

`setq-default`
see Section 12.11.2 [Creating Buffer-Local], page 208,

`unwind-protect`
see Section 11.7 [Nonlocal Exits], page 177,

`while` see Section 11.5 [Iteration], page 174,

Common Lisp note: Here are some comparisons of special forms in GNU Emacs Lisp and Common Lisp. `setq`, `if`, and `catch` are special forms in both Emacs Lisp and Common Lisp. `save-excursion` is a special form in Emacs Lisp, but doesn't exist in Common Lisp. `throw` is a special form in Common Lisp (because it must be able to throw multiple values), but it is a function in Emacs Lisp (which doesn't have multiple values).

10.1.8 Autoloading

The *autoload* feature allows you to call a function or macro whose function definition has not yet been loaded into Emacs. It specifies which file contains the definition. When an autoload object appears as a symbol's function definition, calling that symbol as a function

automatically loads the specified file; then it calls the real definition loaded from that file. The way to arrange for an autoload object to appear as a symbol's function definition is described in Section 16.5 [Autoload], page 305.

10.2 Quoting

The special form `quote` returns its single argument, as written, without evaluating it. This provides a way to include constant symbols and lists, which are not self-evaluating objects, in a program. (It is not necessary to quote self-evaluating objects such as numbers, strings, and vectors.)

`quote` *object* [Special Form]
: This special form returns *object*, without evaluating it. The returned value might be shared and should not be modified. See Section 10.1.1 [Self-Evaluating Forms], page 146.

Because `quote` is used so often in programs, Lisp provides a convenient read syntax for it. An apostrophe character (‘`'`’) followed by a Lisp object (in read syntax) expands to a list whose first element is `quote`, and whose second element is the object. Thus, the read syntax `'x` is an abbreviation for `(quote x)`.

Here are some examples of expressions that use `quote`:

```
(quote (+ 1 2))
     ⇒ (+ 1 2)
(quote foo)
     ⇒ foo
'foo
     ⇒ foo
''foo
     ⇒ 'foo
'(quote foo)
     ⇒ 'foo
['foo]
     ⇒ ['foo]
```

Although the expressions `(list '+ 1 2)` and `'(+ 1 2)` both yield lists equal to `(+ 1 2)`, the former yields a freshly-minted mutable list whereas the latter yields a list built from conses that might be shared and should not be modified. See Section 10.1.1 [Self-Evaluating Forms], page 146.

Other quoting constructs include `function` (see Section 13.7 [Anonymous Functions], page 243), which causes an anonymous lambda expression written in Lisp to be compiled, and ‘`` ` ``’ (see Section 10.3 [Backquote], page 151), which is used to quote only part of a list, while computing and substituting other parts.

10.3 Backquote

Backquote constructs allow you to quote a list, but selectively evaluate elements of that list. In the simplest case, it is identical to the special form `quote`. For example, these two forms yield identical results:

```
`(a list of (+ 2 3) elements)
     ⇒ (a list of (+ 2 3) elements)
'(a list of (+ 2 3) elements)
     ⇒ (a list of (+ 2 3) elements)
```

The special marker ‘,’ inside of the argument to backquote indicates a value that isn't constant. The Emacs Lisp evaluator evaluates the argument of ‘,’, and puts the value in the list structure:

```
`(a list of ,(+ 2 3) elements)
     ⇒ (a list of 5 elements)
```

Substitution with ‘,’ is allowed at deeper levels of the list structure also. For example:

```
`(1 2 (3 ,(+ 4 5)))
     ⇒ (1 2 (3 9))
```

You can also *splice* an evaluated value into the resulting list, using the special marker ‘,@’. The elements of the spliced list become elements at the same level as the other elements of the resulting list. The equivalent code without using ‘`’ is often unreadable. Here are some examples:

```
(setq some-list '(2 3))
     ⇒ (2 3)
(cons 1 (append some-list '(4) some-list))
     ⇒ (1 2 3 4 2 3)
`(1 ,@some-list 4 ,@some-list)
     ⇒ (1 2 3 4 2 3)

(setq list '(hack foo bar))
     ⇒ (hack foo bar)
(cons 'use
  (cons 'the
    (cons 'words (append (cdr list) '(as elements)))))
     ⇒ (use the words foo bar as elements)
`(use the words ,@(cdr list) as elements)
     ⇒ (use the words foo bar as elements)
```

If a subexpression of a backquote construct has no substitutions or splices, it acts like `quote` in that it yields conses, vectors and strings that might be shared and should not be modified. See Section 10.1.1 [Self-Evaluating Forms], page 146.

10.4 Eval

Most often, forms are evaluated automatically, by virtue of their occurrence in a program being run. On rare occasions, you may need to write code that evaluates a form that is computed at run time, such as after reading a form from text being edited or getting one from a property list. On these occasions, use the `eval` function. Often `eval` is not needed and something else should be used instead. For example, to get the value of a variable, while `eval` works, `symbol-value` is preferable; or rather than store expressions in a property list that then need to go through `eval`, it is better to store functions instead that are then passed to `funcall`.

The functions and variables described in this section evaluate forms, specify limits to the evaluation process, or record recently returned values. Loading a file also does evaluation (see Chapter 16 [Loading], page 299).

It is generally cleaner and more flexible to store a function in a data structure, and call it with `funcall` or `apply`, than to store an expression in the data structure and evaluate it. Using functions provides the ability to pass information to them as arguments.

`eval` *form* **&optional** *lexical* [Function]

This is the basic function for evaluating an expression. It evaluates *form* in the current environment, and returns the result. The type of the *form* object determines how it is evaluated. See Section 10.1 [Forms], page 146.

The argument *lexical* specifies the scoping rule for local variables (see Section 12.10 [Variable Scoping], page 201). If it is omitted or `nil`, that means to evaluate *form* using the default dynamic scoping rule. If it is `t`, that means to use the lexical scoping rule. The value of *lexical* can also be a non-empty alist specifying a particular *lexical environment* for lexical bindings; however, this feature is only useful for specialized purposes, such as in Emacs Lisp debuggers. See Section 12.10.3 [Lexical Binding], page 203.

Since `eval` is a function, the argument expression that appears in a call to `eval` is evaluated twice: once as preparation before `eval` is called, and again by the `eval` function itself. Here is an example:

```
(setq foo 'bar)
     ⇒ bar
(setq bar 'baz)
     ⇒ baz
;; Here eval receives argument foo
(eval 'foo)
     ⇒ bar
;; Here eval receives argument bar, which is the value of foo
(eval foo)
     ⇒ baz
```

The number of currently active calls to `eval` is limited to `max-lisp-eval-depth` (see below).

`eval-region` *start end* **&optional** *stream read-function* [Command]

This function evaluates the forms in the current buffer in the region defined by the positions *start* and *end*. It reads forms from the region and calls `eval` on them until the end of the region is reached, or until an error is signaled and not handled.

By default, `eval-region` does not produce any output. However, if *stream* is non-`nil`, any output produced by output functions (see Section 20.5 [Output Functions], page 378), as well as the values that result from evaluating the expressions in the region are printed using *stream*. See Section 20.4 [Output Streams], page 376.

If *read-function* is non-`nil`, it should be a function, which is used instead of `read` to read expressions one by one. This function is called with one argument, the stream for reading input. You can also use the variable `load-read-function` (see [How

Programs Do Loading], page 301) to specify this function, but it is more robust to use the *read-function* argument.

`eval-region` does not move point. It always returns `nil`.

`eval-buffer` **&optional** *buffer-or-name stream filename unibyte print* [Command]

This is similar to `eval-region`, but the arguments provide different optional features. `eval-buffer` operates on the entire accessible portion of buffer *buffer-or-name* (see Section "Narrowing" in *The GNU Emacs Manual*). *buffer-or-name* can be a buffer, a buffer name (a string), or `nil` (or omitted), which means to use the current buffer. *stream* is used as in `eval-region`, unless *stream* is `nil` and *print* non-`nil`. In that case, values that result from evaluating the expressions are still discarded, but the output of the output functions is printed in the echo area. *filename* is the file name to use for `load-history` (see Section 16.9 [Unloading], page 314), and defaults to `buffer-file-name` (see Section 28.4 [Buffer File Name], page 677). If *unibyte* is non-`nil`, `read` converts strings to unibyte whenever possible.

`max-lisp-eval-depth` [User Option]

This variable defines the maximum depth allowed in calls to `eval`, `apply`, and `funcall` before an error is signaled (with error message `"Lisp nesting exceeds max-lisp-eval-depth"`).

This limit, with the associated error when it is exceeded, is how Emacs Lisp avoids infinite recursion on an ill-defined function. If you increase the value of `max-lisp-eval-depth` too much, such code can cause stack overflow instead. On some systems, this overflow can be handled. In that case, normal Lisp evaluation is interrupted and control is transferred back to the top level command loop (`top-level`). Note that there is no way to enter Emacs Lisp debugger in this situation. See Section 19.1.1 [Error Debugging], page 334.

The depth limit counts internal uses of `eval`, `apply`, and `funcall`, such as for calling the functions mentioned in Lisp expressions, and recursive evaluation of function call arguments and function body forms, as well as explicit calls in Lisp code.

The default value of this variable is 1600. If you set it to a value less than 100, Lisp will reset it to 100 if the given value is reached. Entry to the Lisp debugger increases the value, if there is little room left, to make sure the debugger itself has room to execute.

`values` [Variable]

The value of this variable is a list of the values returned by all the expressions that were read, evaluated, and printed from buffers (including the minibuffer) by the standard Emacs commands which do this. (Note that this does *not* include evaluation in `*ielm*` buffers, nor evaluation using `C-j`, `C-x C-e`, and similar evaluation commands in `lisp-interaction-mode`.)

This variable is obsolete, and will be removed in a future version, since it constantly enlarges the memory footprint of the Emacs process. For that reason, we recommend against using it.

The elements of `values` are ordered most recent first.

```
(setq x 1)
     ⇒ 1
(list 'A (1+ 2) auto-save-default)
     ⇒ (A 3 t)
values
     ⇒ ((A 3 t) 1 ...)
```

This variable could be useful for referring back to values of forms recently evaluated. It is generally a bad idea to print the value of `values` itself, since this may be very long. Instead, examine particular elements, like this:

```
;; Refer to the most recent evaluation result.
(nth 0 values)
     ⇒ (A 3 t)
;; That put a new element on,
;;    so all elements move back one.
(nth 1 values)
     ⇒ (A 3 t)
;; This gets the element that was next-to-most-recent
;;    before this example.
(nth 3 values)
     ⇒ 1
```

10.5 Deferred and Lazy Evaluation

Sometimes it is useful to delay the evaluation of an expression, for example if you want to avoid performing a time-consuming calculation if it turns out that the result is not needed in the future of the program. The `thunk` library provides the following functions and macros to support such *deferred evaluation*:

`thunk-delay` *forms...* [Macro]

Return a *thunk* for evaluating the *forms*. A thunk is a closure (see Section 13.10 [Closures], page 250) that inherits the lexical environment of the `thunk-delay` call. Using this macro requires `lexical-binding`.

`thunk-force` *thunk* [Function]

Force *thunk* to perform the evaluation of the forms specified in the `thunk-delay` that created the thunk. The result of the evaluation of the last form is returned. The *thunk* also "remembers" that it has been forced: Any further calls of `thunk-force` with the same *thunk* will just return the same result without evaluating the forms again.

`thunk-let` (*bindings...*) *forms...* [Macro]

This macro is analogous to `let` but creates "lazy" variable bindings. Any binding has the form (*`symbol value-form`*). Unlike `let`, the evaluation of any *value-form* is deferred until the binding of the according *symbol* is used for the first time when evaluating the *forms*. Any *value-form* is evaluated at most once. Using this macro requires `lexical-binding`.

Example:

```
(defun f (number)
  (thunk-let ((derived-number
               (progn (message "Calculating 1 plus 2 times %d" number)
                      (1+ (* 2 number)))))
    (if (> number 10)
        derived-number
      number)))

(f 5)
⇒ 5

(f 12)
⊣ Calculating 1 plus 2 times 12
⇒ 25
```

Because of the special nature of lazily bound variables, it is an error to set them (e.g. with `setq`).

`thunk-let*` (*bindings...*) *forms...* [Macro]

This is like `thunk-let` but any expression in *bindings* is allowed to refer to preceding bindings in this `thunk-let*` form. Using this macro requires `lexical-binding`.

```
(thunk-let* ((x (prog2 (message "Calculating x...")
                    (+ 1 1)
                  (message "Finished calculating x")))
             (y (prog2 (message "Calculating y...")
                    (+ x 1)
                  (message "Finished calculating y")))
             (z (prog2 (message "Calculating z...")
                    (+ y 1)
                  (message "Finished calculating z")))
             (a (prog2 (message "Calculating a...")
                    (+ z 1)
                  (message "Finished calculating a"))))
  (* z x))

⊣ Calculating z...
⊣ Calculating y...
⊣ Calculating x...
⊣ Finished calculating x
⊣ Finished calculating y
⊣ Finished calculating z
⇒ 8
```

`thunk-let` and `thunk-let*` use thunks implicitly: their expansion creates helper symbols and binds them to thunks wrapping the binding expressions. All references to the original variables in the body *forms* are then replaced by an expression that calls `thunk-force`

with the according helper variable as the argument. So, any code using `thunk-let` or `thunk-let*` could be rewritten to use thunks, but in many cases using these macros results in nicer code than using thunks explicitly.

11 Control Structures

A Lisp program consists of a set of *expressions*, or *forms* (see Section 10.1 [Forms], page 146). We control the order of execution of these forms by enclosing them in *control structures*. Control structures are special forms which control when, whether, or how many times to execute the forms they contain.

The simplest order of execution is sequential execution: first form *a*, then form *b*, and so on. This is what happens when you write several forms in succession in the body of a function, or at top level in a file of Lisp code—the forms are executed in the order written. We call this *textual order*. For example, if a function body consists of two forms *a* and *b*, evaluation of the function evaluates first *a* and then *b*. The result of evaluating *b* becomes the value of the function.

Explicit control structures make possible an order of execution other than sequential.

Emacs Lisp provides several kinds of control structure, including other varieties of sequencing, conditionals, iteration, and (controlled) jumps—all discussed below. The built-in control structures are special forms since their subforms are not necessarily evaluated or not evaluated sequentially. You can use macros to define your own control structure constructs (see Chapter 14 [Macros], page 269).

11.1 Sequencing

Evaluating forms in the order they appear is the most common way control passes from one form to another. In some contexts, such as in a function body, this happens automatically. Elsewhere you must use a control structure construct to do this: `progn`, the simplest control construct of Lisp.

A `progn` special form looks like this:

```
(progn a b c ...)
```

and it says to execute the forms *a*, *b*, *c*, and so on, in that order. These forms are called the *body* of the `progn` form. The value of the last form in the body becomes the value of the entire `progn`. `(progn)` returns `nil`.

In the early days of Lisp, `progn` was the only way to execute two or more forms in succession and use the value of the last of them. But programmers found they often needed to use a `progn` in the body of a function, where (at that time) only one form was allowed. So the body of a function was made into an implicit `progn`: several forms are allowed just as in the body of an actual `progn`. Many other control structures likewise contain an implicit `progn`. As a result, `progn` is not used as much as it was many years ago. It is needed now most often inside an `unwind-protect`, `and`, `or`, or in the *then*-part of an `if`.

`progn` *forms...* [Special Form]

This special form evaluates all of the *forms*, in textual order, returning the result of the final form.

```
(progn (print "The first form")
       (print "The second form")
       (print "The third form"))
     ⊣ "The first form"
     ⊣ "The second form"
     ⊣ "The third form"
⇒ "The third form"
```

Two other constructs likewise evaluate a series of forms but return different values:

`prog1` *form1 forms...* [Special Form]

This special form evaluates *form1* and all of the *forms*, in textual order, returning the result of *form1*.

```
(prog1 (print "The first form")
       (print "The second form")
       (print "The third form"))
     ⊣ "The first form"
     ⊣ "The second form"
     ⊣ "The third form"
⇒ "The first form"
```

Here is a way to remove the first element from a list in the variable `x`, then return the value of that former element:

```
(prog1 (car x) (setq x (cdr x)))
```

`prog2` *form1 form2 forms...* [Special Form]

This special form evaluates *form1*, *form2*, and all of the following *forms*, in textual order, returning the result of *form2*.

```
(prog2 (print "The first form")
       (print "The second form")
       (print "The third form"))
     ⊣ "The first form"
     ⊣ "The second form"
     ⊣ "The third form"
⇒ "The second form"
```

11.2 Conditionals

Conditional control structures choose among alternatives. Emacs Lisp has five conditional forms: `if`, which is much the same as in other languages; `when` and `unless`, which are variants of `if`; `cond`, which is a generalized case statement; and `pcase`, which is a generalization of `cond` (see Section 11.4 [Pattern-Matching Conditional], page 163).

`if` *condition then-form else-forms...* [Special Form]

`if` chooses between the *then-form* and the *else-forms* based on the value of *condition*. If the evaluated *condition* is non-`nil`, *then-form* is evaluated and the result returned. Otherwise, the *else-forms* are evaluated in textual order, and the value of the last one is returned. (The *else* part of `if` is an example of an implicit `progn`. See Section 11.1 [Sequencing], page 158.)

If *condition* has the value `nil`, and no *else-forms* are given, `if` returns `nil`.

`if` is a special form because the branch that is not selected is never evaluated—it is ignored. Thus, in this example, `true` is not printed because `print` is never called:

```
(if nil
    (print 'true)
  'very-false)
⇒ very-false
```

`when` *condition then-forms...* [Macro]

This is a variant of `if` where there are no *else-forms*, and possibly several *then-forms*. In particular,

```
(when condition a b c)
```

is entirely equivalent to

```
(if condition (progn a b c) nil)
```

`unless` *condition forms...* [Macro]

This is a variant of `if` where there is no *then-form*:

```
(unless condition a b c)
```

is entirely equivalent to

```
(if condition nil
   a b c)
```

`cond` *clause...* [Special Form]

`cond` chooses among an arbitrary number of alternatives. Each *clause* in the `cond` must be a list. The CAR of this list is the *condition*; the remaining elements, if any, the *body-forms*. Thus, a clause looks like this:

```
(condition body-forms...)
```

`cond` tries the clauses in textual order, by evaluating the *condition* of each clause. If the value of *condition* is non-`nil`, the clause succeeds; then `cond` evaluates its *body-forms*, and returns the value of the last of *body-forms*. Any remaining clauses are ignored.

If the value of *condition* is `nil`, the clause fails, so the `cond` moves on to the following clause, trying its *condition*.

A clause may also look like this:

```
(condition)
```

Then, if *condition* is non-`nil` when tested, the `cond` form returns the value of *condition*.

If every *condition* evaluates to `nil`, so that every clause fails, `cond` returns `nil`.

The following example has four clauses, which test for the cases where the value of `x` is a number, string, buffer and symbol, respectively:

```
(cond ((numberp x) x)
      ((stringp x) x)
      ((bufferp x)
       (setq temporary-hack x) ; multiple body-forms
       (buffer-name x))        ; in one clause
      ((symbolp x) (symbol-value x)))
```

Often we want to execute the last clause whenever none of the previous clauses was successful. To do this, we use `t` as the *condition* of the last clause, like this: `(t body-forms)`. The form `t` evaluates to `t`, which is never `nil`, so this clause never fails, provided the `cond` gets to it at all. For example:

```
(setq a 5)
(cond ((eq a 'hack) 'foo)
      (t "default"))
⇒ "default"
```

This `cond` expression returns `foo` if the value of `a` is `hack`, and returns the string `"default"` otherwise.

Any conditional construct can be expressed with `cond` or with `if`. Therefore, the choice between them is a matter of style. For example:

```
(if a b c)
≡
(cond (a b) (t c))
```

It can be convenient to bind variables in conjunction with using a conditional. It's often the case that you compute a value, and then want to do something with that value if it's non-`nil`. The straightforward way to do that is to just write, for instance:

```
(let ((result1 (do-computation)))
  (when result1
    (let ((result2 (do-more result1)))
      (when result2
        (do-something result2)))))
```

Since this is a very common pattern, Emacs provides a number of macros to make this easier and more readable. The above can be written the following way instead:

```
(when-let ((result1 (do-computation))
           (result2 (do-more result1)))
  (do-something result2))
```

There's a number of variations on this theme, and they're briefly described below.

`if-let` *spec then-form else-forms...* [Macro]

Evaluate each binding in *spec* in turn, like in `let*` (see Section 12.3 [Local Variables], page 190), stopping if a binding value is `nil`. If all are non-`nil`, return the value of *then-form*, otherwise the last form in *else-forms*.

`when-let` *spec then-forms...* [Macro]

Like `if-let`, but without *else-forms*.

`while-let` *spec then-forms...* [Macro]

Like `when-let`, but repeat until a binding in *spec* is `nil`. The return value is always `nil`.

11.3 Constructs for Combining Conditions

This section describes constructs that are often used together with `if` and `cond` to express complicated conditions. The constructs `and` and `or` can also be used individually as kinds of multiple conditional constructs.

`not` *condition* [Function]

This function tests for the falsehood of *condition*. It returns `t` if *condition* is `nil`, and `nil` otherwise. The function `not` is identical to `null`, and we recommend using the name `null` if you are testing for an empty list or `nil` value.

`and` *conditions...* [Special Form]

The `and` special form tests whether all the *conditions* are true. It works by evaluating the *conditions* one by one in the order written.

If any of the *conditions* evaluates to `nil`, then the result of the `and` must be `nil` regardless of the remaining *conditions*; so `and` returns `nil` right away, ignoring the remaining *conditions*.

If all the *conditions* turn out non-`nil`, then the value of the last of them becomes the value of the `and` form. Just `(and)`, with no *conditions*, returns `t`, appropriate because all the *conditions* turned out non-`nil`. (Think about it; which one did not?)

Here is an example. The first condition returns the integer 1, which is not `nil`. Similarly, the second condition returns the integer 2, which is not `nil`. The third condition is `nil`, so the remaining condition is never evaluated.

```
(and (print 1) (print 2) nil (print 3))
     ⊣ 1
     ⊣ 2
⇒ nil
```

Here is a more realistic example of using `and`:

```
(if (and (consp foo) (eq (car foo) 'x))
    (message "foo is a list starting with x"))
```

Note that `(car foo)` is not executed if `(consp foo)` returns `nil`, thus avoiding an error.

`and` expressions can also be written using either `if` or `cond`. Here's how:

```
(and arg1 arg2 arg3)
≡
(if arg1 (if arg2 arg3))
≡
(cond (arg1 (cond (arg2 arg3))))
```

`or` *conditions...* [Special Form]

The `or` special form tests whether at least one of the *conditions* is true. It works by evaluating all the *conditions* one by one in the order written.

If any of the *conditions* evaluates to a non-`nil` value, then the result of the `or` must be non-`nil`; so `or` returns right away, ignoring the remaining *conditions*. The value it returns is the non-`nil` value of the condition just evaluated.

If all the *conditions* turn out `nil`, then the `or` expression returns `nil`. Just `(or)`, with no *conditions*, returns `nil`, appropriate because all the *conditions* turned out `nil`. (Think about it; which one did not?)

For example, this expression tests whether `x` is either `nil` or the integer zero:

```
(or (eq x nil) (eq x 0))
```

Like the `and` construct, `or` can be written in terms of `cond`. For example:

```
(or arg1 arg2 arg3)
≡
(cond (arg1)
      (arg2)
      (arg3))
```

You could almost write `or` in terms of `if`, but not quite:

```
(if arg1 arg1
  (if arg2 arg2
    arg3))
```

This is not completely equivalent because it can evaluate *arg1* or *arg2* twice. By contrast, `(or arg1 arg2 arg3)` never evaluates any argument more than once.

`xor` *condition1 condition2* [Function]

This function returns the boolean exclusive-or of *condition1* and *condition2*. That is, `xor` returns `nil` if either both arguments are `nil`, or both are non-`nil`. Otherwise, it returns the value of that argument which is non-`nil`.

Note that in contrast to `or`, both arguments are always evaluated.

11.4 Pattern-Matching Conditional

Aside from the four basic conditional forms, Emacs Lisp also has a pattern-matching conditional form, the `pcase` macro, a hybrid of `cond` and `cl-case` (see Section "Conditionals" in *Common Lisp Extensions*) that overcomes their limitations and introduces the *pattern matching programming style*. The limitations that `pcase` overcomes are:

- The `cond` form chooses among alternatives by evaluating the predicate *condition* of each of its clauses (see Section 11.2 [Conditionals], page 159). The primary limitation is that variables let-bound in *condition* are not available to the clause's *body-forms*.

  Another annoyance (more an inconvenience than a limitation) is that when a series of *condition* predicates implement equality tests, there is a lot of repeated code. (`cl-case` solves this inconvenience.)

- The `cl-case` macro chooses among alternatives by evaluating the equality of its first argument against a set of specific values.

  Its limitations are two-fold:

  1. The equality tests use `eql`.
  2. The values must be known and written in advance.

  These render `cl-case` unsuitable for strings or compound data structures (e.g., lists or vectors). (`cond` doesn't have these limitations, but it has others, see above.)

Conceptually, the `pcase` macro borrows the first-arg focus of `cl-case` and the clause-processing flow of `cond`, replacing *condition* with a generalization of the equality test which is a variant of *pattern matching*, and adding facilities so that you can concisely express a clause's predicate, and arrange to share let-bindings between a clause's predicate and *body-forms*.

The concise expression of a predicate is known as a *pattern*. When the predicate, called on the value of the first arg, returns non-`nil`, we say that "the pattern matches the value" (or sometimes "the value matches the pattern").

11.4.1 The `pcase` macro

For background, See Section 11.4 [Pattern-Matching Conditional], page 163.

`pcase` *expression* **&rest** *clauses* [Macro]

Each clause in *clauses* has the form: `(`*pattern body-forms...*`)`.

Evaluate *expression* to determine its value, *expval*. Find the first clause in *clauses* whose *pattern* matches *expval* and pass control to that clause's *body-forms*.

If there is a match, the value of `pcase` is the value of the last of *body-forms* in the successful clause. Otherwise, `pcase` evaluates to `nil`.

Each *pattern* has to be a *pcase pattern*, which can use either one of the core patterns defined below, or one of the patterns defined via `pcase-defmacro` (see Section 11.4.2 [Extending pcase], page 170).

The rest of this subsection describes different forms of core patterns, presents some examples, and concludes with important caveats on using the let-binding facility provided by some pattern forms. A core pattern can have the following forms:

`_` (underscore)
: Matches any *expval*. This is also known as *don't care* or *wildcard*.

`'`*val*
: Matches if *expval* equals *val*. The comparison is done as if by `equal` (see Section 2.8 [Equality Predicates], page 34).

keyword
integer
string
: Matches if *expval* equals the literal object. This is a special case of `'`*val*, above, possible because literal objects of these types are self-quoting.

symbol
: Matches any *expval*, and additionally let-binds *symbol* to *expval*, such that this binding is available to *body-forms* (see Section 12.10.1 [Dynamic Binding], page 201).

  If *symbol* is part of a sequencing pattern *seqpat* (e.g., by using `and`, below), the binding is also available to the portion of *seqpat* following the appearance of *symbol*. This usage has some caveats, see [caveats], page 168.

  Two symbols to avoid are `t`, which behaves like `_` (above) and is deprecated, and `nil`, which signals an error. Likewise, it makes no sense to bind keyword symbols (see Section 12.2 [Constant Variables], page 189).

`` `qpat ``
: A backquote-style pattern. See Section 11.4.3 [Backquote Patterns], page 171, for the details.

`(cl-type` *type*`)`
: Matches if *expval* is of type *type*, which is a type descriptor as accepted by `cl-typep` (see Section "Type Predicates" in *Common Lisp Extensions*). Examples:

```
(cl-type integer)
(cl-type (integer 0 10))
```

`(pred` *function*`)`
: Matches if the predicate *function* returns non-`nil` when called on *expval*. The test can be negated with the syntax `(pred (not` *function*`))`. The predicate *function* can have one of the following forms:

  function name (a symbol)
  : Call the named function with one argument, *expval*.

    Example: `integerp`

  lambda expression
  : Call the anonymous function with one argument, *expval* (see Section 13.2 [Lambda Expressions], page 232).

    Example: `(lambda (n) (= 42 n))`

  function call with *n* args
  : Call the function (the first element of the function call) with *n* arguments (the other elements) and an additional *n*+1-th argument that is *expval*.

    Example: `(= 42)`
    In this example, the function is `=`, *n* is one, and the actual function call becomes: `(= 42` *expval*`)`.

`(app` *function pattern*`)`
: Matches if *function* called on *expval* returns a value that matches *pattern*. *function* can take one of the forms described for `pred`, above. Unlike `pred`, however, `app` tests the result against *pattern*, rather than against a boolean truth value.

`(guard` *boolean-expression*`)`
: Matches if *boolean-expression* evaluates to non-`nil`.

`(let` *pattern expr*`)`
: Evaluates *expr* to get *exprval* and matches if *exprval* matches *pattern*. (It is called `let` because *pattern* can bind symbols to values using *symbol*.)

A *sequencing pattern* (also known as *seqpat*) is a pattern that processes its sub-pattern arguments in sequence. There are two for `pcase`: `and` and `or`. They behave in a similar manner to the special forms that share their name (see Section 11.3 [Combining Conditions], page 162), but instead of processing values, they process sub-patterns.

`(and` *pattern1...*`)`
: Attempts to match *pattern1*..., in order, until one of them fails to match. In that case, `and` likewise fails to match, and the rest of the sub-patterns are not tested. If all sub-patterns match, `and` matches.

`(or` *pattern1 pattern2...*`)`

Attempts to match *pattern1*, *pattern2*, ..., in order, until one of them succeeds. In that case, `or` likewise matches, and the rest of the sub-patterns are not tested.

To present a consistent environment to *body-forms* (thus avoiding an evaluation error on match), the set of variables bound by the pattern is the union of the variables bound by each sub-pattern. If a variable is not bound by the sub-pattern that matched, then it is bound to `nil`.

Example: Advantage Over `cl-case`

Here's an example that highlights some advantages `pcase` has over `cl-case` (see Section "Conditionals" in *Common Lisp Extensions*).

```
(pcase (get-return-code x)
  ;; string
  ((and (pred stringp) msg)
   (message "%s" msg))
  ;; symbol
  ('success       (message "Done!"))
  ('would-block   (message "Sorry, can't do it now"))
  ('read-only     (message "The schmilblick is read-only"))
  ('access-denied (message "You do not have the needed rights"))
  ;; default
  (code           (message "Unknown return code %S" code)))
```

With `cl-case`, you would need to explicitly declare a local variable `code` to hold the return value of `get-return-code`. Also `cl-case` is difficult to use with strings because it uses `eql` for comparison.

Example: Using `and`

A common idiom is to write a pattern starting with `and`, with one or more *symbol* sub-patterns providing bindings to the sub-patterns that follow (as well as to the body forms). For example, the following pattern matches single-digit integers.

```
(and
  (pred integerp)
  n                       ; bind n to expval
  (guard (<= -9 n 9)))
```

First, `pred` matches if `(integerp` *expval*`)` evaluates to non-`nil`. Next, `n` is a *symbol* pattern that matches anything and binds `n` to *expval*. Lastly, `guard` matches if the boolean expression `(<= -9 n 9)` (note the reference to `n`) evaluates to non-`nil`. If all these sub-patterns match, `and` matches.

Example: Reformulation with `pcase`

Here is another example that shows how to reformulate a simple matching task from its traditional implementation (function `grok/traditional`) to one using `pcase` (function `grok/pcase`). The docstring for both these functions is: "If OBJ is a string of the form "key:NUMBER", return NUMBER (a string). Otherwise, return the list ("149" default)." First, the traditional implementation (see Section 35.3 [Regular Expressions], page 999):

```
(defun grok/traditional (obj)
  (if (and (stringp obj)
           (string-match "^key:\\([[:digit:]]+\\)$" obj))
      (match-string 1 obj)
    (list "149" 'default)))

(grok/traditional "key:0")   ⇒ "0"
(grok/traditional "key:149") ⇒ "149"
(grok/traditional 'monolith) ⇒ ("149" default)
```

The reformulation demonstrates *symbol* binding as well as `or`, `and`, `pred`, `app` and `let`.

```
(defun grok/pcase (obj)
  (pcase obj
    ((or                                    ; line 1
      (and                                  ; line 2
       (pred stringp)                       ; line 3
       (pred (string-match                  ; line 4
              "^key:\\([[:digit:]]+\\)$"))  ; line 5
       (app (match-string 1)                ; line 6
            val))                           ; line 7
      (let val (list "149" 'default)))      ; line 8
     val)))                                 ; line 9

(grok/pcase "key:0")   ⇒ "0"
(grok/pcase "key:149") ⇒ "149"
(grok/pcase 'monolith) ⇒ ("149" default)
```

The bulk of `grok/pcase` is a single clause of a `pcase` form, the pattern on lines 1-8, the (single) body form on line 9. The pattern is `or`, which tries to match in turn its argument sub-patterns, first `and` (lines 2-7), then `let` (line 8), until one of them succeeds.

As in the previous example (see [Example 1], page 166), `and` begins with a `pred` sub-pattern to ensure the following sub-patterns work with an object of the correct type (string, in this case). If `(stringp` *`expval`*`)` returns `nil`, `pred` fails, and thus `and` fails, too.

The next `pred` (lines 4-5) evaluates `(string-match` `RX` *`expval`*`)` and matches if the result is non-`nil`, which means that *expval* has the desired form: `key:NUMBER`. Again, failing this, `pred` fails and `and`, too.

Lastly (in this series of `and` sub-patterns), `app` evaluates `(match-string 1` *`expval`*`)` (line 6) to get a temporary value *tmp* (i.e., the "NUMBER" substring) and tries to match *tmp* against pattern `val` (line 7). Since that is a *symbol* pattern, it matches unconditionally and additionally binds `val` to *tmp*.

Now that `app` has matched, all `and` sub-patterns have matched, and so `and` matches. Likewise, once `and` has matched, `or` matches and does not proceed to try sub-pattern `let` (line 8).

Let's consider the situation where `obj` is not a string, or it is a string but has the wrong form. In this case, one of the `pred` (lines 3-5) fails to match, thus `and` (line 2) fails to match, thus `or` (line 1) proceeds to try sub-pattern `let` (line 8).

First, `let` evaluates `(list "149" 'default)` to get `("149" default)`, the *exprval*, and then tries to match *exprval* against pattern `val`. Since that is a *symbol* pattern, it matches unconditionally and additionally binds `val` to *exprval*. Now that `let` has matched, `or` matches.

Note how both `and` and `let` sub-patterns finish in the same way: by trying (always successfully) to match against the *symbol* pattern `val`, in the process binding `val`. Thus, `or` always matches and control always passes to the body form (line 9). Because that is the last body form in a successfully matched `pcase` clause, it is the value of `pcase` and likewise the return value of `grok/pcase` (see Section 13.1 [What Is a Function], page 230).

Caveats for *symbol* in Sequencing Patterns

The preceding examples all use sequencing patterns which include the *symbol* sub-pattern in some way. Here are some important details about that usage.

1. When *symbol* occurs more than once in *seqpat*, the second and subsequent occurrences do not expand to re-binding, but instead expand to an equality test using `eq`.

   The following example features a `pcase` form with two clauses and two *seqpat*, A and B. Both A and B first check that *expval* is a pair (using `pred`), and then bind symbols to the `car` and `cdr` of *expval* (using one `app` each).

   For A, because symbol `st` is mentioned twice, the second mention becomes an equality test using `eq`. On the other hand, B uses two separate symbols, `s1` and `s2`, both of which become independent bindings.

   ```
   (defun grok (object)
     (pcase object
       ((and (pred consp)        ; seqpat A
             (app car st)        ; first mention: st
             (app cdr st))       ; second mention: st
        (list 'eq st))
       ((and (pred consp)        ; seqpat B
             (app car s1)        ; first mention: s1
             (app cdr s2))       ; first mention: s2
        (list 'not-eq s1 s2))))

   (let ((s "yow!"))
     (grok (cons s s)))      ⇒ (eq "yow!")
   (grok (cons "yo!" "yo!")) ⇒ (not-eq "yo!" "yo!")
   (grok '(4 2))             ⇒ (not-eq 4 (2))
   ```

2. Side-effecting code referencing *symbol* is undefined. Avoid. For example, here are two similar functions. Both use `and`, *symbol* and `guard`:

   ```
   (defun square-double-digit-p/CLEAN (integer)
     (pcase (* integer integer)
       ((and n (guard (< 9 n 100))) (list 'yes n))
       (sorry (list 'no sorry))))

   (square-double-digit-p/CLEAN 9) ⇒ (yes 81)
   (square-double-digit-p/CLEAN 3) ⇒ (no 9)
   ```

```
(defun square-double-digit-p/MAYBE (integer)
  (pcase (* integer integer)
    ((and n (guard (< 9 (incf n) 100))) (list 'yes n))
    (sorry (list 'no sorry))))

(square-double-digit-p/MAYBE 9) ⇒ (yes 81)
(square-double-digit-p/MAYBE 3) ⇒ (yes 9)  ; WRONG!
```

The difference is in *boolean-expression* in `guard`: `CLEAN` references `n` simply and directly, while `MAYBE` references `n` with a side-effect, in the expression `(incf n)`. When `integer` is 3, here's what happens:

- The first `n` binds it to *expval*, i.e., the result of evaluating `(* 3 3)`, or 9.
- *boolean-expression* is evaluated:

```
start:   (< 9 (incf n)        100)
becomes: (< 9 (setq n (1+ n)) 100)
becomes: (< 9 (setq n (1+ 9)) 100)
becomes: (< 9 (setq n 10)     100)
                                   ; side-effect here!
becomes: (< 9       n         100) ; n now bound to 10
becomes: (< 9      10         100)
becomes: t
```

- Because the result of the evaluation is non-`nil`, `guard` matches, `and` matches, and control passes to that clause's body forms.

Aside from the mathematical incorrectness of asserting that 9 is a double-digit integer, there is another problem with `MAYBE`. The body form references `n` once more, yet we do not see the updated value—10—at all. What happened to it?

To sum up, it's best to avoid side-effecting references to *symbol* patterns entirely, not only in *boolean-expression* (in `guard`), but also in *expr* (in `let`) and *function* (in `pred` and `app`).

3. On match, the clause's body forms can reference the set of symbols the pattern let-binds. When *seqpat* is `and`, this set is the union of all the symbols each of its sub-patterns let-binds. This makes sense because, for `and` to match, all the sub-patterns must match.

   When *seqpat* is `or`, things are different: `or` matches at the first sub-pattern that matches; the rest of the sub-patterns are ignored. It makes no sense for each sub-pattern to let-bind a different set of symbols because the body forms have no way to distinguish which sub-pattern matched and choose among the different sets. For example, the following is invalid:

```
(require 'cl-lib)
(pcase (read-number "Enter an integer: ")
  ((or (and (pred cl-evenp)
            e-num)      ; bind e-num to expval
       o-num)           ; bind o-num to expval
   (list e-num o-num)))
```

```
Enter an integer: 42
error  Symbol's value as variable is void: o-num
Enter an integer: 149
error  Symbol's value as variable is void: e-num
```

Evaluating body form `(list e-num o-num)` signals error. To distinguish between sub-patterns, you can use another symbol, identical in name in all sub-patterns but differing in value. Reworking the above example:

```
(require 'cl-lib)
(pcase (read-number "Enter an integer: ")
  ((and num                                ; line 1
        (or (and (pred cl-evenp)           ; line 2
                 (let spin 'even))         ; line 3
            (let spin 'odd)))              ; line 4
   (list spin num)))                       ; line 5

Enter an integer: 42
⇒ (even 42)
Enter an integer: 149
⇒ (odd 149)
```

Line 1 "factors out" the *expval* binding with `and` and *symbol* (in this case, `num`). On line 2, `or` begins in the same way as before, but instead of binding different symbols, uses `let` twice (lines 3-4) to bind the same symbol `spin` in both sub-patterns. The value of `spin` distinguishes the sub-patterns. The body form references both symbols (line 5).

11.4.2 Extending `pcase`

The `pcase` macro supports several kinds of patterns (see Section 11.4 [Pattern-Matching Conditional], page 163). You can add support for other kinds of patterns using the `pcase-defmacro` macro.

`pcase-defmacro` *name args* [*doc*] **&rest** *body* [Macro]

Define a new kind of pattern for `pcase`, to be invoked as `(`*`name actual-args`*`)`. The `pcase` macro expands this into a function call that evaluates *body*, whose job it is to rewrite the invoked pattern into some other pattern, in an environment where *args* are bound to *actual-args*.

Additionally, arrange to display *doc* along with the docstring of `pcase`. By convention, *doc* should use `EXPVAL` to stand for the result of evaluating *expression* (first arg to `pcase`).

Typically, *body* rewrites the invoked pattern to use more basic patterns. Although all patterns eventually reduce to core patterns, `body` need not use core patterns straight away. The following example defines two patterns, named `less-than` and `integer-less-than`.

```
(pcase-defmacro less-than (n)
  "Matches if EXPVAL is a number less than N."
  `(pred (> ,n)))
```

```
(pcase-defmacro integer-less-than (n)
  "Matches if EXPVAL is an integer less than N."
  `(and (pred integerp)
        (less-than ,n)))
```

Note that the docstrings mention *args* (in this case, only one: `n`) in the usual way, and also mention `EXPVAL` by convention. The first rewrite (i.e., *body* for `less-than`) uses one core pattern: `pred`. The second uses two core patterns: `and` and `pred`, as well as the newly-defined pattern `less-than`. Both use a single backquote construct (see Section 10.3 [Backquote], page 151).

11.4.3 Backquote-Style Patterns

This subsection describes *backquote-style patterns*, a set of builtin patterns that eases structural matching. For background, see Section 11.4 [Pattern-Matching Conditional], page 163.

Backquote-style patterns are a powerful set of `pcase` pattern extensions (created using `pcase-defmacro`) that make it easy to match *expval* against specifications of its *structure*.

For example, to match *expval* that must be a list of two elements whose first element is a specific string and the second element is any value, you can write a core pattern:

```
(and (pred listp)
     ls
     (guard (= 2 (length ls)))
     (guard (string= "first" (car ls)))
     (let second-elem (cadr ls)))
```

or you can write the equivalent backquote-style pattern:

```
`("first" ,second-elem)
```

The backquote-style pattern is more concise, resembles the structure of *expval*, and avoids binding `ls`.

A backquote-style pattern has the form `` `*qpat* `` where *qpat* can have the following forms:

`(`*qpat1* `.` *qpat2*`)`
: Matches if *expval* is a cons cell whose `car` matches *qpat1* and whose `cdr` matches *qpat2*. This readily generalizes to lists as in `(`*qpat1* *qpat2* `...)`.

`[`*qpat1* *qpat2* `...` *qpatm*`]`
: Matches if *expval* is a vector of length *m* whose `0..(`*m*`-1)`th elements match *qpat1*, *qpat2* ... *qpatm*, respectively.

symbol
keyword
number
string
: Matches if the corresponding element of *expval* is `equal` to the specified literal object.

`,`*pattern*
: Matches if the corresponding element of *expval* matches *pattern*. Note that *pattern* is any kind that `pcase` supports. (In the example above, `second-elem` is a *symbol* core pattern; it therefore matches anything, and let-binds `second-elem`.)

The *corresponding element* is the portion of *expval* that is in the same structural position as the structural position of *qpat* in the backquote-style pattern. (In the example above, the corresponding element of `second-elem` is the second element of *expval*.)

Here is an example of using `pcase` to implement a simple interpreter for a little expression language (note that this requires lexical binding for the lambda expression in the `fn` clause to properly capture `body` and `arg` (see Section 12.10.3 [Lexical Binding], page 203):

```
(defun evaluate (form env)
  (pcase form
    (`(add ,x ,y)       (+ (evaluate x env)
                           (evaluate y env)))
    (`(call ,fun ,arg)  (funcall (evaluate fun env)
                                 (evaluate arg env)))
    (`(fn ,arg ,body)   (lambda (val)
                          (evaluate body (cons (cons arg val)
                                               env))))
    ((pred numberp)     form)
    ((pred symbolp)     (cdr (assq form env)))
    (_                  (error "Syntax error: %S" form))))
```

The first three clauses use backquote-style patterns. `` `(add ,x ,y) `` is a pattern that checks that `form` is a three-element list starting with the literal symbol `add`, then extracts the second and third elements and binds them to symbols `x` and `y`, respectively. This is known as *destructuring*, see Section 11.4.4 [Destructuring with pcase Patterns], page 172. The clause body evaluates `x` and `y` and adds the results. Similarly, the `call` clause implements a function call, and the `fn` clause implements an anonymous function definition.

The remaining clauses use core patterns. `(pred numberp)` matches if `form` is a number. On match, the body evaluates it. `(pred symbolp)` matches if `form` is a symbol. On match, the body looks up the symbol in `env` and returns its association. Finally, `_` is the catch-all pattern that matches anything, so it's suitable for reporting syntax errors.

Here are some sample programs in this small language, including their evaluation results:

```
(evaluate '(add 1 2) nil)                  ⇒ 3
(evaluate '(add x y) '((x . 1) (y . 2)))   ⇒ 3
(evaluate '(call (fn x (add 1 x)) 2) nil) ⇒ 3
(evaluate '(sub 1 2) nil)                  ⇒ error
```

11.4.4 Destructuring with `pcase` Patterns

Pcase patterns not only express a condition on the form of the objects they can match, but they can also extract sub-fields of those objects. For example we can extract 2 elements from a list that is the value of the variable `my-list` with the following code:

```
(pcase my-list
  (`(add ,x ,y)  (message "Contains %S and %S" x y)))
```

This will not only extract `x` and `y` but will additionally test that `my-list` is a list containing exactly 3 elements and whose first element is the symbol `add`. If any of those tests fail, `pcase` will immediately return `nil` without calling `message`.

Extraction of multiple values stored in an object is known as *destructuring*. Using `pcase` patterns allows to perform *destructuring binding*, which is similar to a local binding (see

Section 12.3 [Local Variables], page 190), but gives values to multiple elements of a variable by extracting those values from an object of compatible structure.

The macros described in this section use `pcase` patterns to perform destructuring binding. The condition of the object to be of compatible structure means that the object must match the pattern, because only then the object's subfields can be extracted. For example:

```
(pcase-let ((`(add ,x ,y) my-list))
  (message "Contains %S and %S" x y))
```

does the same as the previous example, except that it directly tries to extract `x` and `y` from `my-list` without first verifying if `my-list` is a list which has the right number of elements and has `add` as its first element. The precise behavior when the object does not actually match the pattern is undefined, although the body will not be silently skipped: either an error is signaled or the body is run with some of the variables potentially bound to arbitrary values like `nil`.

The pcase patterns that are useful for destructuring bindings are generally those described in Section 11.4.3 [Backquote Patterns], page 171, since they express a specification of the structure of objects that will match.

For an alternative facility for destructuring binding, see [seq-let], page 114.

`pcase-let` *bindings body...* [Macro]

Perform destructuring binding of variables according to *bindings*, and then evaluate *body*.

bindings is a list of bindings of the form `(`*pattern exp*`)`, where *exp* is an expression to evaluate and *pattern* is a `pcase` pattern.

All *exp*s are evaluated first, after which they are matched against their respective *pattern*, introducing new variable bindings that can then be used inside *body*. The variable bindings are produced by destructuring binding of elements of *pattern* to the values of the corresponding elements of the evaluated *exp*.

Here's a trivial example:

```
(pcase-let ((`(,major ,minor)
     (split-string "image/png" "/")))
  minor)
     ⇒ "png"
```

`pcase-let*` *bindings body...* [Macro]

Perform destructuring binding of variables according to *bindings*, and then evaluate *body*.

bindings is a list of bindings of the form `(`*pattern exp*`)`, where *exp* is an expression to evaluate and *pattern* is a `pcase` pattern. The variable bindings are produced by destructuring binding of elements of *pattern* to the values of the corresponding elements of the evaluated *exp*.

Unlike `pcase-let`, but similarly to `let*`, each *exp* is matched against its corresponding *pattern* before processing the next element of *bindings*, so the variable bindings introduced in each one of the *bindings* are available in the *exp*s of the *bindings* that follow it, additionally to being available in *body*.

`pcase-dolist` (*pattern list*) *body...* [Macro]
Execute *body* once for each element of *list*, on each iteration performing a destructuring binding of variables in *pattern* to the values of the corresponding subfields of the element of *list*. The bindings are performed as if by `pcase-let`. When *pattern* is a simple variable, this ends up being equivalent to `dolist` (see Section 11.5 [Iteration], page 174).

`pcase-setq` *pattern value...* [Macro]
Assign values to variables in a `setq` form, destructuring each *value* according to its respective *pattern*.

`pcase-lambda` *lambda-list* **&rest** *body* [Macro]
This is like `lambda`, but allows each argument to be a pattern. For instance, here's a simple function that takes a cons cell as the argument:

```
(setq fun
      (pcase-lambda (`(,key . ,val))
        (vector key (* val 10))))
(funcall fun '(foo . 2))
    ⇒ [foo 20]
```

11.5 Iteration

Iteration means executing part of a program repetitively. For example, you might want to repeat some computation once for each element of a list, or once for each integer from 0 to *n*. You can do this in Emacs Lisp with the special form `while`:

`while` *condition forms...* [Special Form]
`while` first evaluates *condition*. If the result is non-`nil`, it evaluates *forms* in textual order. Then it reevaluates *condition*, and if the result is non-`nil`, it evaluates *forms* again. This process repeats until *condition* evaluates to `nil`.

There is no limit on the number of iterations that may occur. The loop will continue until either *condition* evaluates to `nil` or until an error or `throw` jumps out of it (see Section 11.7 [Nonlocal Exits], page 177).

The value of a `while` form is always `nil`.

```
(setq num 0)
     ⇒ 0
(while (< num 4)
  (princ (format "Iteration %d." num))
  (setq num (1+ num)))
     ⊣ Iteration 0.
     ⊣ Iteration 1.
     ⊣ Iteration 2.
     ⊣ Iteration 3.
     ⇒ nil
```

To write a repeat-until loop, which will execute something on each iteration and then do the end-test, put the body followed by the end-test in a `progn` as the first argument of `while`, as shown here:

```
(while (progn
         (forward-line 1)
         (not (looking-at "^$"))))
```

This moves forward one line and continues moving by lines until it reaches an empty line. It is peculiar in that the `while` has no body, just the end test (which also does the real work of moving point).

The `dolist` and `dotimes` macros provide convenient ways to write two common kinds of loops.

`dolist` (*var list* [*result*]) *body...* [Macro]

This construct executes *body* once for each element of *list*, binding the variable *var* locally to hold the current element. Then it returns the value of evaluating *result*, or `nil` if *result* is omitted. For example, here is how you could use `dolist` to define the `reverse` function:

```
(defun reverse (list)
  (let (value)
    (dolist (elt list value)
      (setq value (cons elt value)))))
```

`dotimes` (*var count* [*result*]) *body...* [Macro]

This construct executes *body* once for each integer from 0 (inclusive) to *count* (exclusive), binding the variable *var* to the integer for the current iteration. Then it returns the value of evaluating *result*, or `nil` if *result* is omitted. Use of *result* is deprecated. Here is an example of using `dotimes` to do something 100 times:

```
(dotimes (i 100)
  (insert "I will not obey absurd orders\n"))
```

11.6 Generators

A *generator* is a function that produces a potentially-infinite stream of values. Each time the function produces a value, it suspends itself and waits for a caller to request the next value.

`iter-defun` *name args* [*doc*] [*declare*] [*interactive*] *body...* [Macro]

`iter-defun` defines a generator function. A generator function has the same signature as a normal function, but works differently. Instead of executing *body* when called, a generator function returns an iterator object. That iterator runs *body* to generate values, emitting a value and pausing where `iter-yield` or `iter-yield-from` appears. When *body* returns normally, `iter-next` signals `iter-end-of-sequence` with *body*'s result as its condition data.

Any kind of Lisp code is valid inside *body*, but `iter-yield` and `iter-yield-from` cannot appear inside `unwind-protect` forms.

`iter-lambda` *args* [*doc*] [*interactive*] *body...* [Macro]

`iter-lambda` produces an unnamed generator function that works just like a generator function produced with `iter-defun`.

`iter-yield` *value* [Macro]
: When it appears inside a generator function, `iter-yield` indicates that the current iterator should pause and return *value* from `iter-next`. `iter-yield` evaluates to the `value` parameter of next call to `iter-next`.

`iter-yield-from` *iterator* [Macro]
: `iter-yield-from` yields all the values that *iterator* produces and evaluates to the value that *iterator*'s generator function returns normally. While it has control, *iterator* receives values sent to the iterator using `iter-next`.

To use a generator function, first call it normally, producing a *iterator* object. An iterator is a specific instance of a generator. Then use `iter-next` to retrieve values from this iterator. When there are no more values to pull from an iterator, `iter-next` raises an `iter-end-of-sequence` condition with the iterator's final value.

It's important to note that generator function bodies only execute inside calls to `iter-next`. A call to a function defined with `iter-defun` produces an iterator; you must drive this iterator with `iter-next` for anything interesting to happen. Each call to a generator function produces a *different* iterator, each with its own state.

`iter-next` *iterator* **&optional** *value* [Function]
: Retrieve the next value from *iterator*. If there are no more values to be generated (because *iterator*'s generator function returned), `iter-next` signals the `iter-end-of-sequence` condition; the data value associated with this condition is the value with which *iterator*'s generator function returned.

  value is sent into the iterator and becomes the value to which `iter-yield` evaluates. *value* is ignored for the first `iter-next` call to a given iterator, since at the start of *iterator*'s generator function, the generator function is not evaluating any `iter-yield` form.

`iter-close` *iterator* [Function]
: If *iterator* is suspended inside an `unwind-protect`'s `bodyform` and becomes unreachable, Emacs will eventually run unwind handlers after a garbage collection pass. (Note that `iter-yield` is illegal inside an `unwind-protect`'s `unwindforms`.) To ensure that these handlers are run before then, use `iter-close`.

Some convenience functions are provided to make working with iterators easier:

`iter-do` (*var iterator*) *body* ... [Macro]
: Run *body* with *var* bound to each value that *iterator* produces.

The Common Lisp loop facility also contains features for working with iterators. See Section "Loop Facility" in *Common Lisp Extensions*.

The following piece of code demonstrates some important principles of working with iterators.

```
(require 'generator)
(iter-defun my-iter (x)
  (iter-yield (1+ (iter-yield (1+ x))))
   ;; Return normally
```

```
  -1)

(let* ((iter (my-iter 5))
       (iter2 (my-iter 0)))
  ;; Prints 6
  (print (iter-next iter))
  ;; Prints 9
  (print (iter-next iter 8))
  ;; Prints 1; iter and iter2 have distinct states
  (print (iter-next iter2 nil))

  ;; We expect the iter sequence to end now
  (condition-case x
      (iter-next iter)
    (iter-end-of-sequence
      ;; Prints -1, which my-iter returned normally
      (print (cdr x)))))
```

11.7 Nonlocal Exits

A *nonlocal exit* is a transfer of control from one point in a program to another remote point. Nonlocal exits can occur in Emacs Lisp as a result of errors; you can also use them under explicit control. Nonlocal exits unbind all variable bindings made by the constructs being exited.

11.7.1 Explicit Nonlocal Exits: `catch` and `throw`

Most control constructs affect only the flow of control within the construct itself. The function `throw` is the exception to this rule of normal program execution: it performs a nonlocal exit on request. (There are other exceptions, but they are for error handling only.) `throw` is used inside a `catch`, and jumps back to that `catch`. For example:

```
(defun foo-outer ()
  (catch 'foo
    (foo-inner)))

(defun foo-inner ()
  ...
  (if x
      (throw 'foo t))
  ...)
```

The `throw` form, if executed, transfers control straight back to the corresponding `catch`, which returns immediately. The code following the `throw` is not executed. The second argument of `throw` is used as the return value of the `catch`.

The function `throw` finds the matching `catch` based on the first argument: it searches for a `catch` whose first argument is `eq` to the one specified in the `throw`. If there is more than one applicable `catch`, the innermost one takes precedence. Thus, in the above example, the `throw` specifies `foo`, and the `catch` in `foo-outer` specifies the same symbol, so that `catch` is the applicable one (assuming there is no other matching `catch` in between).

Executing `throw` exits all Lisp constructs up to the matching `catch`, including function calls. When binding constructs such as `let` or function calls are exited in this way, the bindings are unbound, just as they are when these constructs exit normally (see Section 12.3 [Local Variables], page 190). Likewise, `throw` restores the buffer and position saved by `save-excursion` (see Section 31.3 [Excursions], page 866), and the narrowing status saved by `save-restriction`. It also runs any cleanups established with the `unwind-protect` special form when it exits that form (see Section 11.7.4 [Cleanups], page 187).

The `throw` need not appear lexically within the `catch` that it jumps to. It can equally well be called from another function called within the `catch`. As long as the `throw` takes place chronologically after entry to the `catch`, and chronologically before exit from it, it has access to that `catch`. This is why `throw` can be used in commands such as `exit-recursive-edit` that throw back to the editor command loop (see Section 22.13 [Recursive Editing], page 476).

> **Common Lisp note:** Most other versions of Lisp, including Common Lisp, have several ways of transferring control nonsequentially: `return`, `return-from`, and `go`, for example. Emacs Lisp has only `throw`. The `cl-lib` library provides versions of some of these. See Section "Blocks and Exits" in *Common Lisp Extensions*.

`catch` *tag body...* [Special Form]

`catch` establishes a return point for the `throw` function. The return point is distinguished from other such return points by *tag*, which may be any Lisp object except `nil`. The argument *tag* is evaluated normally before the return point is established.

With the return point in effect, `catch` evaluates the forms of the *body* in textual order. If the forms execute normally (without error or nonlocal exit) the value of the last body form is returned from the `catch`.

If a `throw` is executed during the execution of *body*, specifying the same value *tag*, the `catch` form exits immediately; the value it returns is whatever was specified as the second argument of `throw`.

`throw` *tag value* [Function]

The purpose of `throw` is to return from a return point previously established with `catch`. The argument *tag* is used to choose among the various existing return points; it must be `eq` to the value specified in the `catch`. If multiple return points match *tag*, the innermost one is used.

The argument *value* is used as the value to return from that `catch`.

If no return point is in effect with tag *tag*, then a `no-catch` error is signaled with data `(tag value)`.

11.7.2 Examples of `catch` and `throw`

One way to use `catch` and `throw` is to exit from a doubly nested loop. (In most languages, this would be done with a `goto`.) Here we compute `(foo i j)` for *i* and *j* varying from 0 to 9:

```
(defun search-foo ()
  (catch 'loop
    (let ((i 0))
      (while (< i 10)
        (let ((j 0))
          (while (< j 10)
            (if (foo i j)
                (throw 'loop (list i j)))
            (setq j (1+ j))))
        (setq i (1+ i))))))
```

If `foo` ever returns non-`nil`, we stop immediately and return a list of *i* and *j*. If `foo` always returns `nil`, the `catch` returns normally, and the value is `nil`, since that is the result of the `while`.

Here are two tricky examples, slightly different, showing two return points at once. First, two return points with the same tag, `hack`:

```
(defun catch2 (tag)
  (catch tag
    (throw 'hack 'yes)))
⇒ catch2

(catch 'hack
  (print (catch2 'hack))
  'no)
⊣ yes
⇒ no
```

Since both return points have tags that match the `throw`, it goes to the inner one, the one established in `catch2`. Therefore, `catch2` returns normally with value `yes`, and this value is printed. Finally the second body form in the outer `catch`, which is `'no`, is evaluated and returned from the outer `catch`.

Now let's change the argument given to `catch2`:

```
(catch 'hack
  (print (catch2 'quux))
  'no)
⇒ yes
```

We still have two return points, but this time only the outer one has the tag `hack`; the inner one has the tag `quux` instead. Therefore, `throw` makes the outer `catch` return the value `yes`. The function `print` is never called, and the body-form `'no` is never evaluated.

11.7.3 Errors

When Emacs Lisp attempts to evaluate a form that, for some reason, cannot be evaluated, it *signals* an *error*.

When an error is signaled, Emacs's default reaction is to print an error message and terminate execution of the current command. This is the right thing to do in most cases, such as if you type `C-f` at the end of the buffer.

In complicated programs, simple termination may not be what you want. For example, the program may have made temporary changes in data structures, or created temporary buffers that should be deleted before the program is finished. In such cases, you would use `unwind-protect` to establish *cleanup expressions* to be evaluated in case of error. (See Section 11.7.4 [Cleanups], page 187.) Occasionally, you may wish the program to continue execution despite an error in a subroutine. In these cases, you would use `condition-case` to establish *error handlers* to recover control in case of error.

For reporting problems without terminating the execution of the current command, consider issuing a warning instead. See Section 41.5 [Warnings], page 1142.

Resist the temptation to use error handling to transfer control from one part of the program to another; use `catch` and `throw` instead. See Section 11.7.1 [Catch and Throw], page 177.

11.7.3.1 How to Signal an Error

Signaling an error means beginning error processing. Error processing normally aborts all or part of the running program and returns to a point that is set up to handle the error (see Section 11.7.3.2 [Processing of Errors], page 181). Here we describe how to signal an error.

Most errors are signaled automatically within Lisp primitives which you call for other purposes, such as if you try to take the CAR of an integer or move forward a character at the end of the buffer. You can also signal errors explicitly with the functions `error` and `signal`.

Quitting, which happens when the user types *C-g*, is not considered an error, but it is handled almost like an error. See Section 22.11 [Quitting], page 472.

Every error specifies an error message, one way or another. The message should state what is wrong ("File does not exist"), not how things ought to be ("File must exist"). The convention in Emacs Lisp is that error messages should start with a capital letter, but should not end with any sort of punctuation.

`error` *format-string* **&rest** *args* [Function]

This function signals an error with an error message constructed by applying `format-message` (see Section 4.7 [Formatting Strings], page 66) to *format-string* and *args*.

These examples show typical uses of `error`:

```
(error "That is an error -- try something else")
     error  That is an error -- try something else

(error "Invalid name `%s'" "A%%B")
     error  Invalid name ‘A%%B’
```

`error` works by calling `signal` with two arguments: the error symbol `error`, and a list containing the string returned by `format-message`.

Typically grave accent and apostrophe in the format translate to matching curved quotes, e.g., `"Missing ‘%s’"` might result in `"Missing ‘foo’"`. See Section 25.4 [Text Quoting Style], page 602, for how to influence or inhibit this translation.

Warning: If you want to use your own string as an error message verbatim, don't just write `(error` *`string`*`)`. If *string string* contains '`%`', '`` ` ``', or '`'` it may be reformatted, with undesirable results. Instead, use `(error "%s"` *`string`*`)`.

When `noninteractive` is non-`nil` (see Section 42.17 [Batch Mode], page 1294), this function kills Emacs if the signaled error has no handler.

`signal` *error-symbol data* [Function]

This function signals an error named by *error-symbol*. The argument *data* is a list of additional Lisp objects relevant to the circumstances of the error.

The argument *error-symbol* must be an *error symbol*—a symbol defined with `define-error`. This is how Emacs Lisp classifies different sorts of errors. See Section 11.7.3.4 [Error Symbols], page 186, for a description of error symbols, error conditions and condition names.

If the error is not handled, the two arguments are used in printing the error message. Normally, this error message is provided by the `error-message` property of *error-symbol*. If *data* is non-`nil`, this is followed by a colon and a comma separated list of the unevaluated elements of *data*. For `error`, the error message is the CAR of *data* (that must be a string). Subcategories of `file-error` are handled specially.

The number and significance of the objects in *data* depends on *error-symbol*. For example, with a `wrong-type-argument` error, there should be two objects in the list: a predicate that describes the type that was expected, and the object that failed to fit that type.

Both *error-symbol* and *data* are available to any error handlers that handle the error: `condition-case` binds a local variable to a list of the form `(error-symbol . data)` (see Section 11.7.3.3 [Handling Errors], page 182).

The function `signal` never returns. If the error *error-symbol* has no handler, and `noninteractive` is non-`nil` (see Section 42.17 [Batch Mode], page 1294), this function eventually kills Emacs.

```
(signal 'wrong-number-of-arguments '(x y))
     error  Wrong number of arguments: x, y

(signal 'no-such-error '("My unknown error condition"))
     error  peculiar error: "My unknown error condition"
```

`user-error` *format-string* **&rest** *args* [Function]

This function behaves exactly like `error`, except that it uses the error symbol `user-error` rather than `error`. As the name suggests, this is intended to report errors on the part of the user, rather than errors in the code itself. For example, if you try to use the command `Info-history-back` (`l`) to move back beyond the start of your Info browsing history, Emacs signals a `user-error`. Such errors do not cause entry to the debugger, even when `debug-on-error` is non-`nil`. See Section 19.1.1 [Error Debugging], page 334.

Common Lisp note: Emacs Lisp has nothing like the Common Lisp concept of continuable errors.

11.7.3.2 How Emacs Processes Errors

When an error is signaled, `signal` searches for an active *handler* for the error. A handler is a sequence of Lisp expressions designated to be executed if an error happens in part

of the Lisp program. If the error has an applicable handler, the handler is executed, and control resumes following the handler. The handler executes in the environment of the `condition-case` that established it; all functions called within that `condition-case` have already been exited, and the handler cannot return to them.

If there is no applicable handler for the error, it terminates the current command and returns control to the editor command loop. (The command loop has an implicit handler for all kinds of errors.) The command loop's handler uses the error symbol and associated data to print an error message. You can use the variable `command-error-function` to control how this is done:

`command-error-function` [Variable]
: This variable, if non-`nil`, specifies a function to use to handle errors that return control to the Emacs command loop. The function should take three arguments: *data*, a list of the same form that `condition-case` would bind to its variable; *context*, a string describing the situation in which the error occurred, or (more often) `nil`; and *caller*, the Lisp function which called the primitive that signaled the error.

An error that has no explicit handler may call the Lisp debugger (see Section 19.1.10 [Invoking the Debugger], page 341). The debugger is enabled if the variable `debug-on-error` (see Section 19.1.1 [Error Debugging], page 334) is non-`nil`. Unlike error handlers, the debugger runs in the environment of the error, so that you can examine values of variables precisely as they were at the time of the error. In batch mode (see Section 42.17 [Batch Mode], page 1294), the Emacs process then normally exits with a non-zero exit status.

11.7.3.3 Writing Code to Handle Errors

The usual effect of signaling an error is to terminate the command that is running and return immediately to the Emacs editor command loop. You can arrange to trap errors occurring in a part of your program by establishing an error handler, with the special form `condition-case`. A simple example looks like this:

```
(condition-case nil
    (delete-file filename)
  (error nil))
```

This deletes the file named *filename*, catching any error and returning `nil` if an error occurs. (You can use the macro `ignore-errors` for a simple case like this; see below.)

The `condition-case` construct is often used to trap errors that are predictable, such as failure to open a file in a call to `insert-file-contents`. It is also used to trap errors that are totally unpredictable, such as when the program evaluates an expression read from the user.

The second argument of `condition-case` is called the *protected form*. (In the example above, the protected form is a call to `delete-file`.) The error handlers go into effect when this form begins execution and are deactivated when this form returns. They remain in effect for all the intervening time. In particular, they are in effect during the execution of functions called by this form, in their subroutines, and so on. This is a good thing, since, strictly speaking, errors can be signaled only by Lisp primitives (including `signal` and `error`) called by the protected form, not by the protected form itself.

The arguments after the protected form are handlers. Each handler lists one or more *condition names* (which are symbols) to specify which errors it will handle. The error symbol specified when an error is signaled also defines a list of condition names. A handler applies to an error if they have any condition names in common. In the example above, there is one handler, and it specifies one condition name, `error`, which covers all errors.

The search for an applicable handler checks all the established handlers starting with the most recently established one. Thus, if two nested `condition-case` forms offer to handle the same error, the inner of the two gets to handle it.

If an error is handled by some `condition-case` form, this ordinarily prevents the debugger from being run, even if `debug-on-error` says this error should invoke the debugger.

If you want to be able to debug errors that are caught by a `condition-case`, set the variable `debug-on-signal` to a non-`nil` value. You can also specify that a particular handler should let the debugger run first, by writing `debug` among the conditions, like this:

```
(condition-case nil
    (delete-file filename)
  ((debug error) nil))
```

The effect of `debug` here is only to prevent `condition-case` from suppressing the call to the debugger. Any given error will invoke the debugger only if `debug-on-error` and the other usual filtering mechanisms say it should. See Section 19.1.1 [Error Debugging], page 334.

`condition-case-unless-debug` *var protected-form handlers...* [Macro]

The macro `condition-case-unless-debug` provides another way to handle debugging of such forms. It behaves exactly like `condition-case`, unless the variable `debug-on-error` is non-`nil`, in which case it does not handle any errors at all.

Once Emacs decides that a certain handler handles the error, it returns control to that handler. To do so, Emacs unbinds all variable bindings made by binding constructs that are being exited, and executes the cleanups of all `unwind-protect` forms that are being exited. Once control arrives at the handler, the body of the handler executes normally.

After execution of the handler body, execution returns from the `condition-case` form. Because the protected form is exited completely before execution of the handler, the handler cannot resume execution at the point of the error, nor can it examine variable bindings that were made within the protected form. All it can do is clean up and proceed.

Error signaling and handling have some resemblance to `throw` and `catch` (see Section 11.7.1 [Catch and Throw], page 177), but they are entirely separate facilities. An error cannot be caught by a `catch`, and a `throw` cannot be handled by an error handler (though using `throw` when there is no suitable `catch` signals an error that can be handled).

`condition-case` *var protected-form handlers...* [Special Form]

This special form establishes the error handlers *handlers* around the execution of *protected-form*. If *protected-form* executes without error, the value it returns becomes the value of the `condition-case` form (in the absence of a success handler; see below). In this case, the `condition-case` has no effect. The `condition-case` form makes a difference when an error occurs during *protected-form*.

Each of the *handlers* is a list of the form `(`*conditions* `body...)`. Here *conditions* is an error condition name to be handled, or a list of condition names (which can

include `debug` to allow the debugger to run before the handler). A condition name of `t` matches any condition. *body* is one or more Lisp expressions to be executed when this handler handles an error. Here are examples of handlers:

```
(error nil)

(arith-error (message "Division by zero"))

((arith-error file-error)
 (message
  "Either division by zero or failure to open a file"))
```

Each error that occurs has an *error symbol* that describes what kind of error it is, and which describes also a list of condition names (see Section 11.7.3.4 [Error Symbols], page 186). Emacs searches all the active `condition-case` forms for a handler that specifies one or more of these condition names; the innermost matching `condition-case` handles the error. Within this `condition-case`, the first applicable handler handles the error.

After executing the body of the handler, the `condition-case` returns normally, using the value of the last form in the handler body as the overall value.

The argument *var* is a variable. `condition-case` does not bind this variable when executing the *protected-form*, only when it handles an error. At that time, it binds *var* locally to an *error description*, which is a list giving the particulars of the error. The error description has the form `(`*error-symbol* `.` *data*`)`. The handler can refer to this list to decide what to do. For example, if the error is for failure opening a file, the file name is the second element of *data*—the third element of the error description.

If *var* is `nil`, that means no variable is bound. Then the error symbol and associated data are not available to the handler.

As a special case, one of the *handlers* can be a list of the form `(:success` *body*`...)`, where *body* is executed with *var* (if non-`nil`) bound to the return value of *protected-form* when that expression terminates without error.

Sometimes it is necessary to re-throw a signal caught by `condition-case`, for some outer-level handler to catch. Here's how to do that:

```
(signal (car err) (cdr err))
```

where `err` is the error description variable, the first argument to `condition-case` whose error condition you want to re-throw. See [Definition of signal], page 181.

`error-message-string` *error-descriptor* [Function]

This function returns the error message string for a given error descriptor. It is useful if you want to handle an error by printing the usual error message for that error. See [Definition of signal], page 181.

Here is an example of using `condition-case` to handle the error that results from dividing by zero. The handler displays the error message (but without a beep), then returns a very large number.

```
(defun safe-divide (dividend divisor)
  (condition-case err
      ;; Protected form.
      (/ dividend divisor)
    ;; The handler.
    (arith-error                        ; Condition.
     ;; Display the usual message for this error.
     (message "%s" (error-message-string err))
     1000000)))
⇒ safe-divide

(safe-divide 5 0)
     ⊣ Arithmetic error: (arith-error)
⇒ 1000000
```

The handler specifies condition name `arith-error` so that it will handle only division-by-zero errors. Other kinds of errors will not be handled (by this `condition-case`). Thus:

```
(safe-divide nil 3)
     error  Wrong type argument: number-or-marker-p, nil
```

Here is a `condition-case` that catches all kinds of errors, including those from `error`:

```
(setq baz 34)
     ⇒ 34

(condition-case err
    (if (eq baz 35)
        t
      ;; This is a call to the function error.
      (error "Rats!  The variable %s was %s, not 35" 'baz baz))
  ;; This is the handler; it is not a form.
  (error (princ (format "The error was: %s" err))
         2))
⊣ The error was: (error "Rats!  The variable baz was 34, not 35")
⇒ 2
```

`ignore-errors` *body...* [Macro]

This construct executes *body*, ignoring any errors that occur during its execution. If the execution is without error, `ignore-errors` returns the value of the last form in *body*; otherwise, it returns `nil`.

Here's the example at the beginning of this subsection rewritten using `ignore-errors`:

```
(ignore-errors
 (delete-file filename))
```

`ignore-error` *condition body...* [Macro]

This macro is like `ignore-errors`, but will only ignore the specific error condition specified.

```
(ignore-error end-of-file
```

```
(read ""))
```

condition can also be a list of error conditions.

`with-demoted-errors` *format body...* [Macro]
This macro is like a milder version of `ignore-errors`. Rather than suppressing errors altogether, it converts them into messages. It uses the string *format* to format the message. *format* should contain a single '`%`'-sequence; e.g., `"Error: %S"`. Use `with-demoted-errors` around code that is not expected to signal errors, but should be robust if one does occur. Note that this macro uses `condition-case-unless-debug` rather than `condition-case`.

11.7.3.4 Error Symbols and Condition Names

When you signal an error, you specify an *error symbol* to specify the kind of error you have in mind. Each error has one and only one error symbol to categorize it. This is the finest classification of errors defined by the Emacs Lisp language.

These narrow classifications are grouped into a hierarchy of wider classes called *error conditions*, identified by *condition names*. The narrowest such classes belong to the error symbols themselves: each error symbol is also a condition name. There are also condition names for more extensive classes, up to the condition name `error` which takes in all kinds of errors (but not `quit`). Thus, each error has one or more condition names: `error`, the error symbol if that is distinct from `error`, and perhaps some intermediate classifications.

`define-error` *name message* **&optional** *parent* [Function]
In order for a symbol to be an error symbol, it must be defined with `define-error` which takes a parent condition (defaults to `error`). This parent defines the conditions that this kind of error belongs to. The transitive set of parents always includes the error symbol itself, and the symbol `error`. Because quitting is not considered an error, the set of parents of `quit` is just `(quit)`.

In addition to its parents, the error symbol has a *message* which is a string to be printed when that error is signaled but not handled. If that message is not valid, the error message '`peculiar error`' is used. See [Definition of signal], page 181.

Internally, the set of parents is stored in the `error-conditions` property of the error symbol and the message is stored in the `error-message` property of the error symbol.

Here is how we define a new error symbol, `new-error`:

```
(define-error 'new-error "A new error" 'my-own-errors)
```

This error has several condition names: `new-error`, the narrowest classification; `my-own-errors`, which we imagine is a wider classification; and all the conditions of `my-own-errors` which should include `error`, which is the widest of all.

The error string should start with a capital letter but it should not end with a period. This is for consistency with the rest of Emacs.

Naturally, Emacs will never signal `new-error` on its own; only an explicit call to `signal` (see [Definition of signal], page 181) in your code can do this:

```
(signal 'new-error '(x y))
     error  A new error: x, y
```

This error can be handled through any of its condition names. This example handles `new-error` and any other errors in the class `my-own-errors`:

```
(condition-case foo
    (bar nil t)
  (my-own-errors nil))
```

The significant way that errors are classified is by their condition names—the names used to match errors with handlers. An error symbol serves only as a convenient way to specify the intended error message and list of condition names. It would be cumbersome to give `signal` a list of condition names rather than one error symbol.

By contrast, using only error symbols without condition names would seriously decrease the power of `condition-case`. Condition names make it possible to categorize errors at various levels of generality when you write an error handler. Using error symbols alone would eliminate all but the narrowest level of classification.

See Appendix F [Standard Errors], page 1395, for a list of the main error symbols and their conditions.

11.7.4 Cleaning Up from Nonlocal Exits

The `unwind-protect` construct is essential whenever you temporarily put a data structure in an inconsistent state; it permits you to make the data consistent again in the event of an error or throw. (Another more specific cleanup construct that is used only for changes in buffer contents is the atomic change group; Section 33.33 [Atomic Changes], page 962.)

`unwind-protect` *body-form cleanup-forms...* [Special Form]

`unwind-protect` executes *body-form* with a guarantee that the *cleanup-forms* will be evaluated if control leaves *body-form*, no matter how that happens. *body-form* may complete normally, or execute a `throw` out of the `unwind-protect`, or cause an error; in all cases, the *cleanup-forms* will be evaluated.

If *body-form* finishes normally, `unwind-protect` returns the value of *body-form*, after it evaluates the *cleanup-forms*. If *body-form* does not finish, `unwind-protect` does not return any value in the normal sense.

Only *body-form* is protected by the `unwind-protect`. If any of the *cleanup-forms* themselves exits nonlocally (via a `throw` or an error), `unwind-protect` is *not* guaranteed to evaluate the rest of them. If the failure of one of the *cleanup-forms* has the potential to cause trouble, then protect it with another `unwind-protect` around that form.

For example, here we make an invisible buffer for temporary use, and make sure to kill it before finishing:

```
(let ((buffer (get-buffer-create " *temp*")))
  (with-current-buffer buffer
    (unwind-protect
        body-form
      (kill-buffer buffer))))
```

You might think that we could just as well write `(kill-buffer (current-buffer))` and dispense with the variable `buffer`. However, the way shown above is safer, if *body-form* happens to get an error after switching to a different buffer! (Alternatively, you could write

a `save-current-buffer` around *body-form*, to ensure that the temporary buffer becomes current again in time to kill it.)

Emacs includes a standard macro called `with-temp-buffer` which expands into more or less the code shown above (see [Current Buffer], page 675). Several of the macros defined in this manual use `unwind-protect` in this way.

Here is an actual example derived from an FTP package. It creates a process (see Chapter 40 [Processes], page 1081) to try to establish a connection to a remote machine. As the function `ftp-login` is highly susceptible to numerous problems that the writer of the function cannot anticipate, it is protected with a form that guarantees deletion of the process in the event of failure. Otherwise, Emacs might fill up with useless subprocesses.

```
(let ((win nil))
  (unwind-protect
      (progn
        (setq process (ftp-setup-buffer host file))
        (if (setq win (ftp-login process host user password))
            (message "Logged in")
          (error "Ftp login failed")))
    (or win (and process (delete-process process)))))
```

This example has a small bug: if the user types *C-g* to quit, and the quit happens immediately after the function `ftp-setup-buffer` returns but before the variable `process` is set, the process will not be killed. There is no easy way to fix this bug, but at least it is very unlikely.

12 Variables

A *variable* is a name used in a program to stand for a value. In Lisp, each variable is represented by a Lisp symbol (see Chapter 9 [Symbols], page 133). The variable name is simply the symbol's name, and the variable's value is stored in the symbol's value cell[1]. See Section 9.1 [Symbol Components], page 133. In Emacs Lisp, the use of a symbol as a variable is independent of its use as a function name.

As previously noted in this manual, a Lisp program is represented primarily by Lisp objects, and only secondarily as text. The textual form of a Lisp program is given by the read syntax of the Lisp objects that constitute the program. Hence, the textual form of a variable in a Lisp program is written using the read syntax for the symbol representing the variable.

12.1 Global Variables

The simplest way to use a variable is *globally*. This means that the variable has just one value at a time, and this value is in effect (at least for the moment) throughout the Lisp system. The value remains in effect until you specify a new one. When a new value replaces the old one, no trace of the old value remains in the variable.

You specify a value for a symbol with `setq`. For example,

```
(setq x '(a b))
```

gives the variable `x` the value `(a b)`. Note that `setq` is a special form (see Section 10.1.7 [Special Forms], page 149); it does not evaluate its first argument, the name of the variable, but it does evaluate the second argument, the new value.

Once the variable has a value, you can refer to it by using the symbol itself as an expression. Thus,

```
x ⇒ (a b)
```

assuming the `setq` form shown above has already been executed.

If you do set the same variable again, the new value replaces the old one:

```
x
     ⇒ (a b)
(setq x 4)
     ⇒ 4
x
     ⇒ 4
```

12.2 Variables that Never Change

In Emacs Lisp, certain symbols normally evaluate to themselves. These include `nil` and `t`, as well as any symbol whose name starts with '`:`' (these are called *keywords*). These symbols cannot be rebound, nor can their values be changed. Any attempt to set or bind `nil` or `t` signals a `setting-constant` error. The same is true for a keyword (a symbol

[1] To be precise, under the default *dynamic scoping* rule, the value cell always holds the variable's current value, but this is not the case under the *lexical scoping* rule. See Section 12.10 [Variable Scoping], page 201, for details.

whose name starts with ':'), if it is interned in the standard obarray, except that setting such a symbol to itself is not an error.

```
nil ≡ 'nil
     ⇒ nil
(setq nil 500)
error  Attempt to set constant symbol: nil
```

`keywordp` *object* [Function]
: function returns `t` if *object* is a symbol whose name starts with ':', interned in the standard obarray, and returns `nil` otherwise.

These constants are fundamentally different from the constants defined using the `defconst` special form (see Section 12.5 [Defining Variables], page 194). A `defconst` form serves to inform human readers that you do not intend to change the value of a variable, but Emacs does not raise an error if you actually change it.

A small number of additional symbols are made read-only for various practical reasons. These include `enable-multibyte-characters`, `most-positive-fixnum`, `most-negative-fixnum`, and a few others. Any attempt to set or bind these also signals a `setting-constant` error.

12.3 Local Variables

Global variables have values that last until explicitly superseded with new values. Sometimes it is useful to give a variable a *local value*—a value that takes effect only within a certain part of a Lisp program. When a variable has a local value, we say that it is *locally bound* to that value, and that it is a *local variable*.

For example, when a function is called, its argument variables receive local values, which are the actual arguments supplied to the function call; these local bindings take effect within the body of the function. To take another example, the `let` special form explicitly establishes local bindings for specific variables, which take effect only within the body of the `let` form.

We also speak of the *global binding*, which is where (conceptually) the global value is kept.

Establishing a local binding saves away the variable's previous value (or lack of one). We say that the previous value is *shadowed*. Both global and local values may be shadowed. If a local binding is in effect, using `setq` on the local variable stores the specified value in the local binding. When that local binding is no longer in effect, the previously shadowed value (or lack of one) comes back.

A variable can have more than one local binding at a time (e.g., if there are nested `let` forms that bind the variable). The *current binding* is the local binding that is actually in effect. It determines the value returned by evaluating the variable symbol, and it is the binding acted on by `setq`.

For most purposes, you can think of the current binding as the innermost local binding, or the global binding if there is no local binding. To be more precise, a rule called the *scoping rule* determines where in a program a local binding takes effect. The default scoping rule in Emacs Lisp is called *dynamic scoping*, which simply states that the current binding

at any given point in the execution of a program is the most recently-created binding for that variable that still exists. For details about dynamic scoping, and an alternative scoping rule called *lexical scoping*, see Section 12.10 [Variable Scoping], page 201. Lately Emacs is moving towards using lexical binding in more and more places, with the goal of eventually making lexical binding the default. In particular, all Emacs Lisp source files and the `*scratch*` buffer use lexical scoping.

The special forms `let` and `let*` exist to create local bindings:

`let` (*bindings...*) *forms...* [Special Form]

This special form sets up local bindings for a certain set of variables, as specified by *bindings*, and then evaluates all of the *forms* in textual order. Its return value is the value of the last form in *forms*. The local bindings set up by `let` will be in effect only within the body of *forms*.

Each of the *bindings* is either (i) a symbol, in which case that symbol is locally bound to `nil`; or (ii) a list of the form `(symbol value-form)`, in which case *symbol* is locally bound to the result of evaluating *value-form*. If *value-form* is omitted, `nil` is used.

All of the *value-form*s in *bindings* are evaluated in the order they appear and *before* binding any of the symbols to them. Here is an example of this: `z` is bound to the old value of `y`, which is 2, not the new value of `y`, which is 1.

```
(setq y 2)
     ⇒ 2

(let ((y 1)
      (z y))
  (list y z))
     ⇒ (1 2)
```

On the other hand, the order of *bindings* is unspecified: in the following example, either 1 or 2 might be printed.

```
(let ((x 1)
      (x 2))
  (print x))
```

Therefore, avoid binding a variable more than once in a single `let` form.

`let*` (*bindings...*) *forms...* [Special Form]

This special form is like `let`, but it binds each variable right after computing its local value, before computing the local value for the next variable. Therefore, an expression in *bindings* can refer to the preceding symbols bound in this `let*` form. Compare the following example with the example above for `let`.

```
(setq y 2)
     ⇒ 2

(let* ((y 1)
       (z y))     ; Use the just-established value of y.
  (list y z))
     ⇒ (1 1)
```

Basically, the `let*` binding of `x` and `y` in the previous example is equivalent to using nested `let` bindings:

```
(let ((y 1))
  (let ((z y))
    (list y z)))
```

`letrec` (*bindings...*) *forms...* [Special Form]

This special form is like `let*`, but all the variables are bound before any of the local values are computed. The values are then assigned to the locally bound variables. This is only useful when lexical binding is in effect, and you want to create closures that refer to bindings that would otherwise not yet be in effect when using `let*`.

For instance, here's a closure that removes itself from a hook after being run once:

```
(letrec ((hookfun (lambda ()
                    (message "Run once")
                    (remove-hook 'post-command-hook hookfun))))
  (add-hook 'post-command-hook hookfun))
```

`dlet` (*bindings...*) *forms...* [Special Form]

This special form is like `let`, but it binds all variables dynamically. This is rarely useful—you usually want to bind normal variables lexically, and special variables (i.e., variables that are defined with `defvar`) dynamically, and this is what `let` does.

`dlet` can be useful when interfacing with old code that assumes that certain variables are dynamically bound (see Section 12.10.1 [Dynamic Binding], page 201), but it's impractical to `defvar` these variables. `dlet` will temporarily make the bound variables special, execute the forms, and then make the variables non-special again.

`named-let` *name bindings* **&rest** *body* [Special Form]

This special form is a looping construct inspired from the Scheme language. It is similar to `let`: It binds the variables in *bindings*, and then evaluates *body*. However, `named-let` also binds *name* to a local function whose formal arguments are the variables in *bindings* and whose body is *body*. This allows *body* to call itself recursively by calling *name*, where the arguments passed to *name* are used as the new values of the bound variables in the recursive invocation.

Example of a loop summing a list of numbers:

```
(named-let sum ((numbers '(1 2 3 4))
                (running-sum 0))
  (if numbers
      (sum (cdr numbers) (+ running-sum (car numbers)))
    running-sum))
⇒ 10
```

Recursive calls to *name* that occur in *tail positions* in *body* are guaranteed to be optimized as *tail calls*, which means that they will not consume any additional stack space no matter how deeply the recursion runs. Such recursive calls will effectively jump to the top of the loop with new values for the variables.

A function call is in the tail position if it's the very last thing done so that the value returned by the call is the value of *body* itself, as is the case in the recursive call to `sum` above.

Warning: `named-let` works as expected only when lexical-binding is enabled. See Section 12.10.3 [Lexical Binding], page 203.

Here is a complete list of the other facilities that create local bindings:

- Function calls (see Chapter 13 [Functions], page 230).
- Macro calls (see Chapter 14 [Macros], page 269).
- `condition-case` (see Section 11.7.3 [Errors], page 179).

Variables can also have buffer-local bindings (see Section 12.11 [Buffer-Local Variables], page 207); a few variables have terminal-local bindings (see Section 30.2 [Multiple Terminals], page 789). These kinds of bindings work somewhat like ordinary local bindings, but they are localized depending on where you are in Emacs.

12.4 When a Variable is Void

We say that a variable is void if its symbol has an unassigned value cell (see Section 9.1 [Symbol Components], page 133).

Under Emacs Lisp's default dynamic scoping rule (see Section 12.10 [Variable Scoping], page 201), the value cell stores the variable's current (local or global) value. Note that an unassigned value cell is *not* the same as having `nil` in the value cell. The symbol `nil` is a Lisp object and can be the value of a variable, just as any other object can be; but it is still a value. If a variable is void, trying to evaluate the variable signals a `void-variable` error, instead of returning a value.

Under the optional lexical scoping rule, the value cell only holds the variable's global value—the value outside of any lexical binding construct. When a variable is lexically bound, the local value is determined by the lexical environment; hence, variables can have local values even if their symbols' value cells are unassigned.

`makunbound` *symbol* [Function]

This function empties out the value cell of *symbol*, making the variable void. It returns *symbol*.

If *symbol* has a dynamic local binding, `makunbound` voids the current binding, and this voidness lasts only as long as the local binding is in effect. Afterwards, the previously shadowed local or global binding is reexposed; then the variable will no longer be void, unless the reexposed binding is void too.

Here are some examples (assuming dynamic binding is in effect):

```
(setq x 1)                ; Put a value in the global binding.
     ⇒ 1
(let ((x 2))              ; Locally bind it.
  (makunbound 'x)         ; Void the local binding.
  x)
error  Symbol's value as variable is void: x
x                         ; The global binding is unchanged.
     ⇒ 1

(let ((x 2))              ; Locally bind it.
  (let ((x 3))            ; And again.
    (makunbound 'x)       ; Void the innermost-local binding.
    x))                   ; And refer: it's void.
error  Symbol's value as variable is void: x
```

```
(let ((x 2))
  (let ((x 3))
    (makunbound 'x))      ; Void inner binding, then remove it.
  x)                      ; Now outer let binding is visible.
     ⇒ 2
```

`boundp` *variable* [Function]

This function returns `t` if *variable* (a symbol) is not void, and `nil` if it is void.

Here are some examples (assuming dynamic binding is in effect):

```
(boundp 'abracadabra)          ; Starts out void.
     ⇒ nil
(let ((abracadabra 5))         ; Locally bind it.
  (boundp 'abracadabra))
     ⇒ t
(boundp 'abracadabra)          ; Still globally void.
     ⇒ nil
(setq abracadabra 5)           ; Make it globally nonvoid.
     ⇒ 5
(boundp 'abracadabra)
     ⇒ t
```

12.5 Defining Global Variables

A *variable definition* is a construct that announces your intention to use a symbol as a global variable. It uses the special forms `defvar` or `defconst`, which are documented below.

A variable definition serves three purposes. First, it informs people who read the code that the symbol is *intended* to be used a certain way (as a variable). Second, it informs the Lisp system of this, optionally supplying an initial value and a documentation string. Third, it provides information to programming tools such as `etags`, allowing them to find where the variable was defined.

The difference between `defconst` and `defvar` is mainly a matter of intent, serving to inform human readers of whether the value should ever change. Emacs Lisp does not actually prevent you from changing the value of a variable defined with `defconst`. One notable difference between the two forms is that `defconst` unconditionally initializes the variable, whereas `defvar` initializes it only if it is originally void.

To define a customizable variable, you should use `defcustom` (which calls `defvar` as a subroutine). See Section 15.3 [Variable Definitions], page 281.

`defvar` *symbol* [*value* [*doc-string*]] [Special Form]

This special form defines *symbol* as a variable. Note that *symbol* is not evaluated; the symbol to be defined should appear explicitly in the `defvar` form. The variable is marked as *special*, meaning that it should always be dynamically bound (see Section 12.10 [Variable Scoping], page 201).

If *value* is specified, and *symbol* is void (i.e., it has no dynamically bound value; see Section 12.4 [Void Variables], page 193), then *value* is evaluated and *symbol* is set to the result. But if *symbol* is not void, *value* is not evaluated, and *symbol*'s value is left unchanged. If *value* is omitted, the value of *symbol* is not changed in any case.

Note that specifying a value, even `nil`, marks the variable as special permanently. Whereas if *value* is omitted then the variable is only marked special locally (i.e. within

the current lexical scope, or file if at the top-level). This can be useful for suppressing byte compilation warnings, see Section 17.6 [Compiler Errors], page 322.

If *symbol* has a buffer-local binding in the current buffer, `defvar` acts on the default value, which is buffer-independent, rather than the buffer-local binding. It sets the default value if the default value is void. See Section 12.11 [Buffer-Local Variables], page 207.

If *symbol* is already let bound (e.g., if the `defvar` form occurs in a `let` form), then `defvar` sets the toplevel default value, like `set-default-toplevel-value`. The let binding remains in effect until its binding construct exits. See Section 12.10 [Variable Scoping], page 201.

When you evaluate a top-level `defvar` form with *C-M-x* (`eval-defun`) or with *C-x C-e* (`eval-last-sexp`) in Emacs Lisp mode, a special feature of these two commands arranges to set the variable unconditionally, without testing whether its value is void.

If the *doc-string* argument is supplied, it specifies the documentation string for the variable (stored in the symbol's `variable-documentation` property). See Chapter 25 [Documentation], page 597.

Here are some examples. This form defines `foo` but does not initialize it:

```
(defvar foo)
     ⇒ foo
```

This example initializes the value of `bar` to `23`, and gives it a documentation string:

```
(defvar bar 23
  "The normal weight of a bar.")
     ⇒ bar
```

The `defvar` form returns *symbol*, but it is normally used at top level in a file where its value does not matter.

For a more elaborate example of using `defvar` without a value, see [Local defvar example], page 205.

`defconst` *symbol value* [*doc-string*] [Special Form]

This special form defines *symbol* as a value and initializes it. It informs a person reading your code that *symbol* has a standard global value, established here, that should not be changed by the user or by other programs. Note that *symbol* is not evaluated; the symbol to be defined must appear explicitly in the `defconst`.

The `defconst` form, like `defvar`, marks the variable as *special*, meaning that it should always be dynamically bound (see Section 12.10 [Variable Scoping], page 201). In addition, it marks the variable as risky (see Section 12.12 [File Local Variables], page 213).

`defconst` always evaluates *value*, and sets the value of *symbol* to the result. If *symbol* does have a buffer-local binding in the current buffer, `defconst` sets the default value, not the buffer-local value. (But you should not be making buffer-local bindings for a symbol that is defined with `defconst`.)

An example of the use of `defconst` is Emacs's definition of `float-pi`—the mathematical constant *pi*, which ought not to be changed by anyone (attempts by the

Indiana State Legislature notwithstanding). As the second form illustrates, however, `defconst` is only advisory.

```
(defconst float-pi 3.141592653589793 "The value of Pi.")
     ⇒ float-pi
(setq float-pi 3)
     ⇒ float-pi
float-pi
     ⇒ 3
```

Warning: If you use a `defconst` or `defvar` special form while the variable has a local binding (made with `let`, or a function argument), it sets the local binding rather than the global binding. This is not what you usually want. To prevent this, use these special forms at top level in a file, where normally no local binding is in effect, and make sure to load the file before making a local binding for the variable.

12.6 Tips for Defining Variables Robustly

When you define a variable whose value is a function, or a list of functions, use a name that ends in '`-function`' or '`-functions`', respectively.

There are several other variable name conventions; here is a complete list:

'`...-hook`'
: The variable is a normal hook (see Section 24.1 [Hooks], page 525).

'`...-function`'
: The value is a function.

'`...-functions`'
: The value is a list of functions.

'`...-form`'
: The value is a form (an expression).

'`...-forms`'
: The value is a list of forms (expressions).

'`...-predicate`'
: The value is a predicate—a function of one argument that returns non-`nil` for success and `nil` for failure.

'`...-flag`'
: The value is significant only as to whether it is `nil` or not. Since such variables often end up acquiring more values over time, this convention is not strongly recommended.

'`...-program`'
: The value is a program name.

'`...-command`'
: The value is a whole shell command.

'`...-switches`'
: The value specifies options for a command.

‘*prefix*--...’
: The variable is intended for internal use and is defined in the file *prefix*.el. (Emacs code contributed before 2018 may follow other conventions, which are being phased out.)

‘...-internal’
: The variable is intended for internal use and is defined in C code. (Emacs code contributed before 2018 may follow other conventions, which are being phased out.)

When you define a variable, always consider whether you should mark it as safe or risky; see Section 12.12 [File Local Variables], page 213.

When defining and initializing a variable that holds a complicated value (such as a syntax table for a major mode), it's best to put the entire computation of the value into the `defvar`, like this:

```
(defvar my-major-mode-syntax-table
  (let ((table (make-syntax-table)))
    (modify-syntax-entry ?# "<" table)
    ...
    table)
  docstring)
```

This method has several benefits. First, if the user quits while loading the file, the variable is either still uninitialized or initialized properly, never in-between. If it is still uninitialized, reloading the file will initialize it properly. Second, reloading the file once the variable is initialized will not alter it; that is important if the user has changed its value. Third, evaluating the `defvar` form with *C-M-x* will reinitialize the variable completely.

12.7 Accessing Variable Values

The usual way to reference a variable is to write the symbol which names it. See Section 10.1.2 [Symbol Forms], page 146.

Occasionally, you may want to reference a variable which is only determined at run time. In that case, you cannot specify the variable name in the text of the program. You can use the `symbol-value` function to extract the value.

`symbol-value` *symbol* [Function]
: This function returns the value stored in *symbol*'s value cell. This is where the variable's current (dynamic) value is stored. If the variable has no local binding, this is simply its global value. If the variable is void, a `void-variable` error is signaled.

  If the variable is lexically bound, the value reported by `symbol-value` is not necessarily the same as the variable's lexical value, which is determined by the lexical environment rather than the symbol's value cell. See Section 12.10 [Variable Scoping], page 201.

```
(setq abracadabra 5)
     ⇒ 5
(setq foo 9)
     ⇒ 9
```

```
;; Here the symbol abracadabra
;;    is the symbol whose value is examined.
(let ((abracadabra 'foo))
  (symbol-value 'abracadabra))
     ⇒ foo

;; Here, the value of abracadabra,
;;    which is foo,
;;    is the symbol whose value is examined.
(let ((abracadabra 'foo))
  (symbol-value abracadabra))
     ⇒ 9

(symbol-value 'abracadabra)
     ⇒ 5
```

12.8 Setting Variable Values

The usual way to change the value of a variable is with the special form `setq`. When you need to compute the choice of variable at run time, use the function `set`.

`setq` [*symbol form*]. . . [Special Form]

This special form is the most common method of changing a variable's value. Each *symbol* is given a new value, which is the result of evaluating the corresponding *form*. The current binding of the symbol is changed.

`setq` does not evaluate *symbol*; it sets the symbol that you write. We say that this argument is *automatically quoted*. The '`q`' in `setq` stands for "quoted".

The value of the `setq` form is the value of the last *form*.

```
(setq x (1+ 2))
     ⇒ 3
x                    ; x now has a global value.
     ⇒ 3
(let ((x 5))
  (setq x 6)         ; The local binding of x is set.
  x)
     ⇒ 6
x                    ; The global value is unchanged.
     ⇒ 3
```

Note that the first *form* is evaluated, then the first *symbol* is set, then the second *form* is evaluated, then the second *symbol* is set, and so on:

```
(setq x 10           ; Notice that x is set before
      y (1+ x))      ;    the value of y is computed.
     ⇒ 11
```

`set` *symbol value* [Function]

This function puts *value* in the value cell of *symbol*. Since it is a function rather than a special form, the expression written for *symbol* is evaluated to obtain the symbol to set. The return value is *value*.

When dynamic variable binding is in effect (the default), `set` has the same effect as `setq`, apart from the fact that `set` evaluates its *symbol* argument whereas `setq` does not. But when a variable is lexically bound, `set` affects its *dynamic* value, whereas `setq` affects its current (lexical) value. See Section 12.10 [Variable Scoping], page 201.

```
(set one 1)
error  Symbol's value as variable is void: one
(set 'one 1)
     ⇒ 1
(set 'two 'one)
     ⇒ one
(set two 2)          ; two evaluates to symbol one.
     ⇒ 2
one                  ; So it is one that was set.
     ⇒ 2
(let ((one 1))       ; This binding of one is set,
  (set 'one 3)       ;    not the global value.
  one)
     ⇒ 3
one
     ⇒ 2
```

If *symbol* is not actually a symbol, a `wrong-type-argument` error is signaled.

```
(set '(x y) 'z)
error  Wrong type argument: symbolp, (x y)
```

`setopt` [*symbol form*]... [Macro]

This is like `setq` (see above), but meant for user options. This macro uses the Customize machinery to set the variable(s) (see Section 15.3 [Variable Definitions], page 281). In particular, `setopt` will run the setter function associated with the variable. For instance, if you have:

```
(defcustom my-var 1
  "My var."
  :type 'number
  :set (lambda (var val)
         (set-default var val)
         (message "We set %s to %s" var val)))
```

then the following, in addition to setting `my-var` to '2', will also issue a message:

```
(setopt my-var 2)
```

`setopt` also checks whether the value is valid for the user option. For instance, using `setopt` to set a user option defined with a `number` type to a string will signal an error.

Unlike `defcustom` and related customization commands, such as `customize-variable`, `setopt` is meant for non-interactive use, in particular in the user init file. For that reason, it doesn't record the standard, saved, and

user-set values, and doesn't mark the variable as candidate for saving in the custom file.

The `setopt` macro can be used on regular, non-user option variables, but is much less efficient than `setq`. The main use case for this macro is setting user options in the user's init file.

12.9 Running a function when a variable is changed.

It is sometimes useful to take some action when a variable changes its value. The *variable watchpoint* facility provides the means to do so. Some possible uses for this feature include keeping display in sync with variable settings, and invoking the debugger to track down unexpected changes to variables (see Section 19.1.5 [Variable Debugging], page 338).

The following functions may be used to manipulate and query the watch functions for a variable.

`add-variable-watcher` *symbol watch-function* [Function]

This function arranges for *watch-function* to be called whenever *symbol* is modified. Modifications through aliases (see Section 12.15 [Variable Aliases], page 222) will have the same effect.

watch-function will be called, just before changing the value of *symbol*, with 4 arguments: *symbol*, *newval*, *operation*, and *where*. *symbol* is the variable being changed. *newval* is the value it will be changed to. (The old value is available to *watch-function* as the value of *symbol*, since it was not yet changed to *newval*.) *operation* is a symbol representing the kind of change, one of: `set`, `let`, `unlet`, `makunbound`, or `defvaralias`. *where* is a buffer if the buffer-local value of the variable is being changed, `nil` otherwise.

`remove-variable-watcher` *symbol watch-function* [Function]

This function removes *watch-function* from *symbol*'s list of watchers.

`get-variable-watchers` *symbol* [Function]

This function returns the list of *symbol*'s active watcher functions.

12.9.1 Limitations

There are a couple of ways in which a variable could be modified (or at least appear to be modified) without triggering a watchpoint.

Since watchpoints are attached to symbols, modification to the objects contained within variables (e.g., by a list modification function see Section 5.6 [Modifying Lists], page 87) is not caught by this mechanism.

Additionally, C code can modify the value of variables directly, bypassing the watchpoint mechanism.

A minor limitation of this feature, again because it targets symbols, is that only variables of dynamic scope may be watched. This poses little difficulty, since modifications to lexical variables can be discovered easily by inspecting the code within the scope of the variable (unlike dynamic variables, which can be modified by any code at all, see Section 12.10 [Variable Scoping], page 201).

12.10 Scoping Rules for Variable Bindings

When you create a local binding for a variable, that binding takes effect only within a limited portion of the program (see Section 12.3 [Local Variables], page 190). This section describes exactly what this means.

Each local binding has a certain *scope* and *extent*. *Scope* refers to *where* in the textual source code the binding can be accessed. *Extent* refers to *when*, as the program is executing, the binding exists.

By default, the local bindings that Emacs creates are *dynamic bindings*. Such a binding has *dynamic scope*, meaning that any part of the program can potentially access the variable binding. It also has *dynamic extent*, meaning that the binding lasts only while the binding construct (such as the body of a `let` form) is being executed.

Emacs can optionally create *lexical bindings*. A lexical binding has *lexical scope*, meaning that any reference to the variable must be located textually within the binding construct[2]. It also has *indefinite extent*, meaning that under some circumstances the binding can live on even after the binding construct has finished executing, by means of special objects called *closures*.

The dynamic binding was (and still is) the default in Emacs for many years, but lately Emacs is moving towards using lexical binding in more and more places, with the goal of eventually making that the default.

The following subsections describe dynamic binding and lexical binding in greater detail, and how to enable lexical binding in Emacs Lisp programs.

12.10.1 Dynamic Binding

By default, the local variable bindings made by Emacs are dynamic bindings. When a variable is dynamically bound, its current binding at any point in the execution of the Lisp program is simply the most recently-created dynamic local binding for that symbol, or the global binding if there is no such local binding.

Dynamic bindings have dynamic scope and extent, as shown by the following example:

```
(defvar x -99)  ; x receives an initial value of -99.

(defun getx ()
  x)            ; x is used free in this function.

(let ((x 1))    ; x is dynamically bound.
  (getx))
     ⇒ 1

;; After the let form finishes, x reverts to its
;; previous value, which is -99.

(getx)
     ⇒ -99
```

[2] With some exceptions; for instance, a lexical binding can also be accessed from the Lisp debugger.

The function `getx` refers to `x`. This is a *free* reference, in the sense that there is no binding for `x` within that `defun` construct itself. When we call `getx` from within a `let` form in which `x` is (dynamically) bound, it retrieves the local value (i.e., 1). But when we call `getx` outside the `let` form, it retrieves the global value (i.e., −99).

Here is another example, which illustrates setting a dynamically bound variable using `setq`:

```
(defvar x -99)        ; x receives an initial value of −99.

(defun addx ()
  (setq x (1+ x)))    ; Add 1 to x and return its new value.

(let ((x 1))
  (addx)
  (addx))
     ⇒ 3              ; The two addx calls add to x twice.

;; After the let form finishes, x reverts to its
;; previous value, which is −99.

(addx)
     ⇒ -98
```

Dynamic binding is implemented in Emacs Lisp in a simple way. Each symbol has a value cell, which specifies its current dynamic value (or absence of value). See Section 9.1 [Symbol Components], page 133. When a symbol is given a dynamic local binding, Emacs records the contents of the value cell (or absence thereof) in a stack, and stores the new local value in the value cell. When the binding construct finishes executing, Emacs pops the old value off the stack, and puts it in the value cell.

Note that when code using Dynamic Binding is native compiled the native compiler will not perform any Lisp specific optimization.

12.10.2 Proper Use of Dynamic Binding

Dynamic binding is a powerful feature, as it allows programs to refer to variables that are not defined within their local textual scope. However, if used without restraint, this can also make programs hard to understand. There are two clean ways to use this technique:

- If a variable has no global definition, use it as a local variable only within a binding construct, such as the body of the `let` form where the variable was bound. If this convention is followed consistently throughout a program, the value of the variable will not affect, nor be affected by, any uses of the same variable symbol elsewhere in the program.
- Otherwise, define the variable with `defvar`, `defconst` (see Section 12.5 [Defining Variables], page 194), or `defcustom` (see Section 15.3 [Variable Definitions], page 281). Usually, the definition should be at top-level in an Emacs Lisp file. As far as possible, it should include a documentation string which explains the meaning and purpose of the variable. You should also choose the variable's name to avoid name conflicts (see Section D.1 [Coding Conventions], page 1336).

Then you can bind the variable anywhere in a program, knowing reliably what the effect will be. Wherever you encounter the variable, it will be easy to refer back to the definition, e.g., via the `C-h v` command (provided the variable definition has been loaded into Emacs). See Section "Name Help" in *The GNU Emacs Manual*.

For example, it is common to use local bindings for customizable variables like `case-fold-search`:

```
(defun search-for-abc ()
  "Search for the string \"abc\", ignoring case differences."
  (let ((case-fold-search t))
    (re-search-forward "abc")))
```

12.10.3 Lexical Binding

Lexical binding was introduced to Emacs, as an optional feature, in version 24.1. We expect its importance to increase with time. Lexical binding opens up many more opportunities for optimization, so programs using it are likely to run faster in future Emacs versions. Lexical binding is also more compatible with concurrency, which was added to Emacs in version 26.1.

A lexically-bound variable has *lexical scope*, meaning that any reference to the variable must be located textually within the binding construct. Here is an example (see the next subsection, for how to actually enable lexical binding):

```
(let ((x 1))     ; x is lexically bound.
  (+ x 3))
     ⇒ 4

(defun getx ()
  x)             ; x is used free in this function.

(let ((x 1))     ; x is lexically bound.
  (getx))
error  Symbol's value as variable is void: x
```

Here, the variable `x` has no global value. When it is lexically bound within a `let` form, it can be used in the textual confines of that `let` form. But it can *not* be used from within a `getx` function called from the `let` form, since the function definition of `getx` occurs outside the `let` form itself.

Here is how lexical binding works. Each binding construct defines a *lexical environment*, specifying the variables that are bound within the construct and their local values. When the Lisp evaluator wants the current value of a variable, it looks first in the lexical environment; if the variable is not specified in there, it looks in the symbol's value cell, where the dynamic value is stored.

(Internally, the lexical environment is a list whose members are usually cons cells that are symbol-value pairs, but some of its members can be symbols rather than cons cells. A symbol in the list means the lexical environment declared that symbol's variable as locally considered to be dynamically bound. This list can be passed as the second argument to the `eval` function, in order to specify a lexical environment in which to evaluate a form.

See Section 10.4 [Eval], page 152. Most Emacs Lisp programs, however, should not interact directly with lexical environments in this way; only specialized programs like debuggers.)

Lexical bindings have indefinite extent. Even after a binding construct has finished executing, its lexical environment can be "kept around" in Lisp objects called *closures*. A closure is created when you define a named or anonymous function with lexical binding enabled. See Section 13.10 [Closures], page 250, for details.

When a closure is called as a function, any lexical variable references within its definition use the retained lexical environment. Here is an example:

```
(defvar my-ticker nil)    ; We will use this dynamically bound
                          ; variable to store a closure.

(let ((x 0))              ; x is lexically bound.
  (setq my-ticker (lambda ()
                    (setq x (1+ x)))))
    ⇒ (closure ((x . 0)) ()
        (setq x (1+ x)))

(funcall my-ticker)
    ⇒ 1

(funcall my-ticker)
    ⇒ 2

(funcall my-ticker)
    ⇒ 3

x                         ; Note that x has no global value.
error  Symbol's value as variable is void: x
```

The `let` binding defines a lexical environment in which the variable `x` is locally bound to 0. Within this binding construct, we define a lambda expression which increments `x` by one and returns the incremented value. This lambda expression is automatically turned into a closure, in which the lexical environment lives on even after the `let` binding construct has exited. Each time we evaluate the closure, it increments `x`, using the binding of `x` in that lexical environment.

Note that unlike dynamic variables which are tied to the symbol object itself, the relationship between lexical variables and symbols is only present in the interpreter (or compiler). Therefore, functions which take a symbol argument (like `symbol-value`, `boundp`, and `set`) can only retrieve or modify a variable's dynamic binding (i.e., the contents of its symbol's value cell).

12.10.4 Using Lexical Binding

When loading an Emacs Lisp file or evaluating a Lisp buffer, lexical binding is enabled if the buffer-local variable `lexical-binding` is non-`nil`:

`lexical-binding` [Variable]

If this buffer-local variable is non-`nil`, Emacs Lisp files and buffers are evaluated using lexical binding instead of dynamic binding. (However, special variables are still dynamically bound; see below.) If `nil`, dynamic binding is used for all local variables. This variable is typically set for a whole Emacs Lisp file, as a file local variable (see Section 12.12 [File Local Variables], page 213). Note that unlike other such variables, this one must be set in the first line of a file.

When evaluating Emacs Lisp code directly using an `eval` call, lexical binding is enabled if the *lexical* argument to `eval` is non-`nil`. See Section 10.4 [Eval], page 152.

Lexical binding is also enabled in Lisp Interaction and IELM mode, used in the `*scratch*` and `*ielm*` buffers, and also when evaluating expressions via `M-:` (`eval-expression`) and when processing the `--eval` command-line options of Emacs (see Section "Action Arguments" in *The GNU Emacs Manual*) and `emacsclient` (see Section "emacsclient Options" in *The GNU Emacs Manual*).

Even when lexical binding is enabled, certain variables will continue to be dynamically bound. These are called *special variables*. Every variable that has been defined with `defvar`, `defcustom` or `defconst` is a special variable (see Section 12.5 [Defining Variables], page 194). All other variables are subject to lexical binding.

Using `defvar` without a value, it is possible to bind a variable dynamically just in one file, or in just one part of a file while still binding it lexically elsewhere. For example:

```
(let (_)
  (defvar x)       ; Let-bindings of x will be dynamic within this let.
  (let ((x -99))   ; This is a dynamic binding of x.
    (defun get-dynamic-x ()
      x)))

(let ((x 'lexical)) ; This is a lexical binding of x.
  (defun get-lexical-x ()
    x))

(let (_)
  (defvar x)
  (let ((x 'dynamic))
    (list (get-lexical-x)
          (get-dynamic-x))))
    ⇒ (lexical dynamic)
```

`special-variable-p` *symbol* [Function]

This function returns non-`nil` if *symbol* is a special variable (i.e., it has a `defvar`, `defcustom`, or `defconst` variable definition). Otherwise, the return value is `nil`.

Note that since this is a function, it can only return non-`nil` for variables which are permanently special, but not for those that are only special in the current lexical scope.

The use of a special variable as a formal argument in a function is not supported.

12.10.5 Converting to Lexical Binding

Converting an Emacs Lisp program to lexical binding is easy. First, add a file-local variable setting of `lexical-binding` to `t` in the header line of the Emacs Lisp source file (see Section 12.12 [File Local Variables], page 213). Second, check that every variable in the program which needs to be dynamically bound has a variable definition, so that it is not inadvertently bound lexically.

A simple way to find out which variables need a variable definition is to byte-compile the source file. See Chapter 17 [Byte Compilation], page 317. If a non-special variable is used outside of a `let` form, the byte-compiler will warn about reference or assignment to a free variable. If a non-special variable is bound but not used within a `let` form, the byte-compiler will warn about an unused lexical variable. The byte-compiler will also issue a warning if you use a special variable as a function argument.

A warning about a reference or an assignment to a free variable is usually a clear sign that that variable should be marked as dynamically scoped, so you need to add an appropriate `defvar` before the first use of that variable.

A warning about an unused variable may be a good hint that the variable was intended to be dynamically scoped (because it is actually used, but in another function), but it may also be an indication that the variable is simply really not used and could simply be removed. So you need to find out which case it is, and based on that, either add a `defvar` or remove the variable altogether. If removal is not possible or not desirable (typically because it is a formal argument and that we cannot or don't want to change all the callers), you can also add a leading underscore to the variable's name to indicate to the compiler that this is a variable known not to be used.)

Cross-file variable checking

Caution: This is an experimental feature that may change or disappear without prior notice.

The byte-compiler can also warn about lexical variables that are special in other Emacs Lisp files, often indicating a missing `defvar` declaration. This useful but somewhat specialized check requires three steps:

1. Byte-compile all files whose special variable declarations may be of interest, with the environment variable `EMACS_GENERATE_DYNVARS` set to a nonempty string. These are typically all the files in the same package or related packages or Emacs subsystems. The process will generate a file whose name ends in `.dynvars` for each compiled Emacs Lisp file.
2. Concatenate the `.dynvars` files into a single file.
3. Byte-compile the files that need to be checked, this time with the environment variable `EMACS_DYNVARS_FILE` set to the name of the aggregated file created in step 2.

Here is an example illustrating how this could be done, assuming that a Unix shell and `make` are used for byte-compilation:

```
$ rm *.elc                                  # force recompilation
$ EMACS_GENERATE_DYNVARS=1 make             # generate .dynvars
$ cat *.dynvars > ~/my-dynvars              # combine .dynvars
$ rm *.elc                                  # force recompilation
$ EMACS_DYNVARS_FILE=~/my-dynvars make      # perform checks
```

12.11 Buffer-Local Variables

Global and local variable bindings are found in most programming languages in one form or another. Emacs, however, also supports additional, unusual kinds of variable binding, such as *buffer-local* bindings, which apply only in one buffer. Having different values for a variable in different buffers is an important customization method. (Variables can also have bindings that are local to each terminal. See Section 30.2 [Multiple Terminals], page 789.)

12.11.1 Introduction to Buffer-Local Variables

A buffer-local variable has a buffer-local binding associated with a particular buffer. The binding is in effect when that buffer is current; otherwise, it is not in effect. If you set the variable while a buffer-local binding is in effect, the new value goes in that binding, so its other bindings are unchanged. This means that the change is visible only in the buffer where you made it.

The variable's ordinary binding, which is not associated with any specific buffer, is called the *default binding*. In most cases, this is the global binding.

A variable can have buffer-local bindings in some buffers but not in other buffers. The default binding is shared by all the buffers that don't have their own bindings for the variable. (This includes all newly-created buffers.) If you set the variable in a buffer that does not have a buffer-local binding for it, this sets the default binding, so the new value is visible in all the buffers that see the default binding.

The most common use of buffer-local bindings is for major modes to change variables that control the behavior of commands. For example, C mode and Lisp mode both set the variable `paragraph-start` to specify that only blank lines separate paragraphs. They do this by making the variable buffer-local in the buffer that is being put into C mode or Lisp mode, and then setting it to the new value for that mode. See Section 24.2 [Major Modes], page 527.

The usual way to make a buffer-local binding is with `make-local-variable`, which is what major mode commands typically use. This affects just the current buffer; all other buffers (including those yet to be created) will continue to share the default value unless they are explicitly given their own buffer-local bindings.

A more powerful operation is to mark the variable as *automatically buffer-local* by calling `make-variable-buffer-local`. You can think of this as making the variable local in all buffers, even those yet to be created. More precisely, the effect is that setting the variable automatically makes the variable local to the current buffer if it is not already so. All buffers start out by sharing the default value of the variable as usual, but setting the variable creates a buffer-local binding for the current buffer. The new value is stored in the buffer-local binding, leaving the default binding untouched. This means that the default value cannot be changed with `setq` in any buffer; the only way to change it is with `setq-default`.

Warning: When a variable has buffer-local bindings in one or more buffers, `let` rebinds the binding that's currently in effect. For instance, if the current buffer has a buffer-local value, `let` temporarily rebinds that. If no buffer-local bindings are in effect, `let` rebinds the default value. If inside the `let` you then change to a different current buffer in which a different binding is in effect, you won't see the `let` binding any more. And if you exit the

`let` while still in the other buffer, you won't see the unbinding occur (though it will occur properly). Here is an example to illustrate:

```
(setq foo 'g)
(set-buffer "a")
(make-local-variable 'foo)
(setq foo 'a)
(let ((foo 'temp))
  ;; foo ⇒ 'temp  ; let binding in buffer ‘a’
  (set-buffer "b")
  ;; foo ⇒ 'g     ; the global value since foo is not local in ‘b’
  body...)
foo ⇒ 'g         ; exiting restored the local value in buffer ‘a’,
                 ; but we don't see that in buffer ‘b’
(set-buffer "a") ; verify the local value was restored
foo ⇒ 'a
```

Note that references to `foo` in *body* access the buffer-local binding of buffer ‘`b`’.

When a file specifies local variable values, these become buffer-local values when you visit the file. See Section "File Variables" in *The GNU Emacs Manual*.

A buffer-local variable cannot be made terminal-local (see Section 30.2 [Multiple Terminals], page 789).

12.11.2 Creating and Deleting Buffer-Local Bindings

`make-local-variable` *variable* [Command]

This function creates a buffer-local binding in the current buffer for *variable* (a symbol). Other buffers are not affected. The value returned is *variable*.

The buffer-local value of *variable* starts out as the same value *variable* previously had. If *variable* was void, it remains void.

```
;; In buffer ‘b1’:
(setq foo 5)                ; Affects all buffers.
     ⇒ 5
(make-local-variable 'foo)  ; Now it is local in ‘b1’.
     ⇒ foo
foo                         ; That did not change
     ⇒ 5                    ;   the value.
(setq foo 6)                ; Change the value
     ⇒ 6                    ;   in ‘b1’.
foo
     ⇒ 6

;; In buffer ‘b2’, the value hasn't changed.
(with-current-buffer "b2"
  foo)
     ⇒ 5
```

Making a variable buffer-local within a `let`-binding for that variable does not work reliably, unless the buffer in which you do this is not current either on entry to or

exit from the `let`. This is because `let` does not distinguish between different kinds of bindings; it knows only which variable the binding was made for.

It is an error to make a constant or a read-only variable buffer-local. See Section 12.2 [Constant Variables], page 189.

If the variable is terminal-local (see Section 30.2 [Multiple Terminals], page 789), this function signals an error. Such variables cannot have buffer-local bindings as well.

Warning: do not use `make-local-variable` for a hook variable. The hook variables are automatically made buffer-local as needed if you use the *local* argument to `add-hook` or `remove-hook`.

`setq-local` **&rest** *pairs* [Macro]

pairs is a list of variable and value pairs. This macro creates a buffer-local binding in the current buffer for each of the variables, and gives them a buffer-local value. It is equivalent to calling `make-local-variable` followed by `setq` for each of the variables. The variables should be unquoted symbols.

```
(setq-local var1 "value1"
            var2 "value2")
```

`make-variable-buffer-local` *variable* [Command]

This function marks *variable* (a symbol) automatically buffer-local, so that any subsequent attempt to set it will make it local to the current buffer at the time. Unlike `make-local-variable`, with which it is often confused, this cannot be undone, and affects the behavior of the variable in all buffers.

A peculiar wrinkle of this feature is that binding the variable (with `let` or other binding constructs) does not create a buffer-local binding for it. Only setting the variable (with `set` or `setq`), while the variable does not have a `let`-style binding that was made in the current buffer, does so.

If *variable* does not have a default value, then calling this command will give it a default value of `nil`. If *variable* already has a default value, that value remains unchanged. Subsequently calling `makunbound` on *variable* will result in a void buffer-local value and leave the default value unaffected.

The value returned is *variable*.

It is an error to make a constant or a read-only variable buffer-local. See Section 12.2 [Constant Variables], page 189.

Warning: Don't assume that you should use `make-variable-buffer-local` for user-option variables, simply because users *might* want to customize them differently in different buffers. Users can make any variable local, when they wish to. It is better to leave the choice to them.

The time to use `make-variable-buffer-local` is when it is crucial that no two buffers ever share the same binding. For example, when a variable is used for internal purposes in a Lisp program which depends on having separate values in separate buffers, then using `make-variable-buffer-local` can be the best solution.

`defvar-local` *variable value* **&optional** *docstring* [Macro]

This macro defines *variable* as a variable with initial value *value* and *docstring*, and marks it as automatically buffer-local. It is equivalent to calling `defvar` followed by `make-variable-buffer-local`. *variable* should be an unquoted symbol.

`local-variable-p` *variable* **&optional** *buffer* [Function]
This returns `t` if *variable* is buffer-local in buffer *buffer* (which defaults to the current buffer); otherwise, `nil`.

`local-variable-if-set-p` *variable* **&optional** *buffer* [Function]
This returns `t` if *variable* either has a buffer-local value in buffer *buffer*, or is automatically buffer-local. Otherwise, it returns `nil`. If omitted or `nil`, *buffer* defaults to the current buffer.

`buffer-local-value` *variable buffer* [Function]
This function returns the buffer-local binding of *variable* (a symbol) in buffer *buffer*. If *variable* does not have a buffer-local binding in buffer *buffer*, it returns the default value (see Section 12.11.3 [Default Value], page 211) of *variable* instead.

`buffer-local-boundp` *variable buffer* [Function]
This returns non-`nil` if there's either a buffer-local binding of *variable* (a symbol) in buffer *buffer*, or *variable* has a global binding.

`buffer-local-variables` **&optional** *buffer* [Function]
This function returns a list describing the buffer-local variables in buffer *buffer*. (If *buffer* is omitted, the current buffer is used.) Normally, each list element has the form `(`*sym* `.` *val*`)`, where *sym* is a buffer-local variable (a symbol) and *val* is its buffer-local value. But when a variable's buffer-local binding in *buffer* is void, its list element is just *sym*.

```
(make-local-variable 'foobar)
(makunbound 'foobar)
(make-local-variable 'bind-me)
(setq bind-me 69)
(setq lcl (buffer-local-variables))
    ;; First, built-in variables local in all buffers:
⇒ ((mark-active . nil)
    (buffer-undo-list . nil)
    (mode-name . "Fundamental")
    ...
    ;; Next, non-built-in buffer-local variables.
    ;; This one is buffer-local and void:
    foobar
    ;; This one is buffer-local and nonvoid:
    (bind-me . 69))
```

Note that storing new values into the CDRs of cons cells in this list does *not* change the buffer-local values of the variables.

`kill-local-variable` *variable* [Command]
This function deletes the buffer-local binding (if any) for *variable* (a symbol) in the current buffer. As a result, the default binding of *variable* becomes visible in this buffer. This typically results in a change in the value of *variable*, since the default value is usually different from the buffer-local value just eliminated.

If you kill the buffer-local binding of a variable that automatically becomes buffer-local when set, this makes the default value visible in the current buffer. However, if you set the variable again, that will once again create a buffer-local binding for it.

`kill-local-variable` returns *variable*.

This function is a command because it is sometimes useful to kill one buffer-local variable interactively, just as it is useful to create buffer-local variables interactively.

`kill-all-local-variables` **&optional** *kill-permanent* [Function]

This function eliminates all the buffer-local variable bindings of the current buffer. As a result, the buffer will see the default values of most variables. By default, for variables marked as permanent and local hook functions that have a non-`nil` `permanent-local-hook` property (see Section 24.1.2 [Setting Hooks], page 526) won't be killed, but if the optional *kill-permanent* argument is non-`nil`, even these variables will be killed.

This function also resets certain other information pertaining to the buffer: it sets the local keymap to `nil`, the syntax table to the value of `(standard-syntax-table)`, the case table to `(standard-case-table)`, and the abbrev table to the value of `fundamental-mode-abbrev-table`.

The very first thing this function does is run the normal hook `change-major-mode-hook` (see below).

Every major mode command begins by calling this function, which has the effect of switching to Fundamental mode and erasing most of the effects of the previous major mode. To ensure that this does its job, the variables that major modes set should not be marked permanent.

`kill-all-local-variables` returns `nil`.

`change-major-mode-hook` [Variable]

The function `kill-all-local-variables` runs this normal hook before it does anything else. This gives major modes a way to arrange for something special to be done if the user switches to a different major mode. It is also useful for buffer-specific minor modes that should be forgotten if the user changes the major mode.

For best results, make this variable buffer-local, so that it will disappear after doing its job and will not interfere with the subsequent major mode. See Section 24.1 [Hooks], page 525.

A buffer-local variable is *permanent* if the variable name (a symbol) has a `permanent-local` property that is non-`nil`. Such variables are unaffected by `kill-all-local-variables`, and their local bindings are therefore not cleared by changing major modes. Permanent locals are appropriate for data pertaining to where the file came from or how to save it, rather than with how to edit the contents.

12.11.3 The Default Value of a Buffer-Local Variable

The global value of a variable with buffer-local bindings is also called the *default* value, because it is the value that is in effect whenever neither the current buffer nor the selected frame has its own binding for the variable.

The functions `default-value` and `setq-default` access and change a variable's default value regardless of whether the current buffer has a buffer-local binding. For example, you could use `setq-default` to change the default setting of `paragraph-start` for most buffers; and this would work even when you are in a C or Lisp mode buffer that has a buffer-local value for this variable.

The special forms `defvar` and `defconst` also set the default value (if they set the variable at all), rather than any buffer-local value.

`default-value` *symbol* [Function]
: This function returns *symbol*'s default value. This is the value that is seen in buffers and frames that do not have their own values for this variable. If *symbol* is not buffer-local, this is equivalent to `symbol-value` (see Section 12.7 [Accessing Variables], page 197).

`default-boundp` *symbol* [Function]
: The function `default-boundp` tells you whether *symbol*'s default value is nonvoid. If `(default-boundp 'foo)` returns `nil`, then `(default-value 'foo)` would get an error.

  `default-boundp` is to `default-value` as `boundp` is to `symbol-value`.

`setq-default` [*symbol form*]... [Special Form]
: This special form gives each *symbol* a new default value, which is the result of evaluating the corresponding *form*. It does not evaluate *symbol*, but does evaluate *form*. The value of the `setq-default` form is the value of the last *form*.

  If a *symbol* is not buffer-local for the current buffer, and is not marked automatically buffer-local, `setq-default` has the same effect as `setq`. If *symbol* is buffer-local for the current buffer, then this changes the value that other buffers will see (as long as they don't have a buffer-local value), but not the value that the current buffer sees.

```
;; In buffer 'foo':
(make-local-variable 'buffer-local)
     ⇒ buffer-local
(setq buffer-local 'value-in-foo)
     ⇒ value-in-foo
(setq-default buffer-local 'new-default)
     ⇒ new-default
buffer-local
     ⇒ value-in-foo
(default-value 'buffer-local)
     ⇒ new-default

;; In (the new) buffer 'bar':
buffer-local
     ⇒ new-default
(default-value 'buffer-local)
     ⇒ new-default
(setq buffer-local 'another-default)
     ⇒ another-default
```

```
(default-value 'buffer-local)
     ⇒ another-default

;; Back in buffer 'foo':
buffer-local
     ⇒ value-in-foo
(default-value 'buffer-local)
     ⇒ another-default
```

`set-default` *symbol value* [Function]

This function is like `setq-default`, except that *symbol* is an ordinary evaluated argument.

```
(set-default (car '(a b c)) 23)
     ⇒ 23
(default-value 'a)
     ⇒ 23
```

A variable can be let-bound (see Section 12.3 [Local Variables], page 190) to a value. This makes its global value shadowed by the binding; `default-value` will then return the value from that binding, not the global value, and `set-default` will be prevented from setting the global value (it will change the let-bound value instead). The following two functions allow to reference the global value even if it's shadowed by a let-binding.

`default-toplevel-value` *symbol* [Function]

This function returns the *top-level* default value of *symbol*, which is its value outside of any let-binding.

```
(defvar variable 'global-value)
    ⇒ variable
(let ((variable 'let-binding))
  (default-value 'variable))
    ⇒ let-binding
(let ((variable 'let-binding))
  (default-toplevel-value 'variable))
    ⇒ global-value
```

`set-default-toplevel-value` *symbol value* [Function]

This function sets the top-level default value of *symbol* to the specified *value*. This comes in handy when you want to set the global value of *symbol* regardless of whether your code runs in the context of *symbol*'s let-binding.

12.12 File Local Variables

A file can specify local variable values; Emacs uses these to create buffer-local bindings for those variables in the buffer visiting that file. See Section "Local Variables in Files" in *The GNU Emacs Manual*, for basic information about file-local variables. This section describes the functions and variables that affect how file-local variables are processed.

If a file-local variable could specify an arbitrary function or Lisp expression that would be called later, visiting a file could take over your Emacs. Emacs protects against this by

automatically setting only those file-local variables whose specified values are known to be safe. Other file-local variables are set only if the user agrees.

For additional safety, `read-circle` is temporarily bound to `nil` when Emacs reads file-local variables (see Section 20.3 [Input Functions], page 375). This prevents the Lisp reader from recognizing circular and shared Lisp structures (see Section 2.6 [Circular Objects], page 30).

`enable-local-variables` [User Option]

This variable controls whether to process file-local variables. The possible values are:

`t` (the default)
: Set the safe variables, and query (once) about any unsafe variables.

`:safe`
: Set only the safe variables and do not query.

`:all`
: Set all the variables and do not query.

`nil`
: Don't set any variables.

anything else
: Query (once) about all the variables.

`inhibit-local-variables-regexps` [Variable]

This is a list of regular expressions. If a file has a name matching an element of this list, then it is not scanned for any form of file-local variable. For examples of why you might want to use this, see Section 24.2.2 [Auto Major Mode], page 532.

`permanently-enabled-local-variables` [Variable]

Some local variable settings will, by default, be heeded even if `enable-local-variables` is `nil`. By default, this is only the case for the `lexical-binding` local variable setting, but this can be controlled by using this variable, which is a list of symbols.

`hack-local-variables` **&optional** *handle-mode* [Function]

This function parses, and binds or evaluates as appropriate, any local variables specified by the contents of the current buffer. The variable `enable-local-variables` has its effect here. However, this function does not look for the '`mode:`' local variable in the '`-*-`' line. `set-auto-mode` does that, also taking `enable-local-variables` into account (see Section 24.2.2 [Auto Major Mode], page 532).

This function works by walking the alist stored in `file-local-variables-alist` and applying each local variable in turn. It calls `before-hack-local-variables-hook` and `hack-local-variables-hook` before and after applying the variables, respectively. It only calls the before-hook if the alist is non-`nil`; it always calls the other hook. This function ignores a '`mode`' element if it specifies the same major mode as the buffer already has.

If the optional argument *handle-mode* is `t`, then all this function does is return a symbol specifying the major mode, if the '`-*-`' line or the local variables list specifies one, and `nil` otherwise. It does not set the mode or any other file-local variable. If *handle-mode* has any value other than `nil` or `t`, any settings of '`mode`' in the '`-*-`' line or the local variables list are ignored, and the other settings are applied. If *handle-mode* is `nil`, all the file local variables are set.

`file-local-variables-alist` [Variable]
: This buffer-local variable holds the alist of file-local variable settings. Each element of the alist is of the form `(var . value)`, where *var* is a symbol of the local variable and *value* is its value. When Emacs visits a file, it first collects all the file-local variables into this alist, and then the `hack-local-variables` function applies them one by one.

`before-hack-local-variables-hook` [Variable]
: Emacs calls this hook immediately before applying file-local variables stored in `file-local-variables-alist`.

`hack-local-variables-hook` [Variable]
: Emacs calls this hook immediately after it finishes applying file-local variables stored in `file-local-variables-alist`.

You can specify safe values for a variable with a `safe-local-variable` property. The property has to be a function of one argument; any value is safe if the function returns non-`nil` given that value. Many commonly-encountered file variables have `safe-local-variable` properties; these include `fill-column`, `fill-prefix`, and `indent-tabs-mode`. For boolean-valued variables that are safe, use `booleanp` as the property value.

If you want to define `safe-local-variable` properties for variables defined in C source code, add the names and the properties of those variables to the list in the "Safe local variables" section of `files.el`.

When defining a user option using `defcustom`, you can set its `safe-local-variable` property by adding the arguments `:safe` *function* to `defcustom` (see Section 15.3 [Variable Definitions], page 281). However, a safety predicate defined using `:safe` will only be known once the package that contains the `defcustom` is loaded, which is often too late. As an alternative, you can use the autoload cookie (see Section 16.5 [Autoload], page 305) to assign the option its safety predicate, like this:

```
;;;###autoload (put 'var 'safe-local-variable 'pred)
```

The safe value definitions specified with `autoload` are copied into the package's autoloads file (`loaddefs.el` for most packages bundled with Emacs), and are known to Emacs since the beginning of a session.

`safe-local-variable-values` [User Option]
: This variable provides another way to mark some variable values as safe. It is a list of cons cells `(var . val)`, where *var* is a variable name and *val* is a value which is safe for that variable.

  When Emacs asks the user whether or not to obey a set of file-local variable specifications, the user can choose to mark them as safe. Doing so adds those variable/value pairs to `safe-local-variable-values`, and saves it to the user's custom file.

`ignored-local-variable-values` [User Option]
: If there are some values of particular local variables that you always want to ignore completely, you can use this variable. Its value has the same form as `safe-local-variable-values`; a file-local variable setting to the value that appears in the list will always be ignored when processing the local variables specified by the file. As with

that variable, when Emacs queries the user about whether to obey file-local variables, the user can choose to ignore their particular values permanently, and that will alter this variable and save it to the user's custom file. Variable-value pairs that appear in this variable take precedence over the same pairs in `safe-local-variable-values`.

`safe-local-variable-p` *sym val* [Function]
This function returns non-`nil` if it is safe to give *sym* the value *val*, based on the above criteria.

Some variables are considered *risky*. If a variable is risky, it is never entered automatically into `safe-local-variable-values`; Emacs always queries before setting a risky variable, unless the user explicitly allows a value by customizing `safe-local-variable-values` directly.

Any variable whose name has a non-`nil` `risky-local-variable` property is considered risky. When you define a user option using `defcustom`, you can set its `risky-local-variable` property by adding the arguments `:risky` *value* to `defcustom` (see Section 15.3 [Variable Definitions], page 281). In addition, any variable whose name ends in any of '`-command`', '`-frame-alist`', '`-function`', '`-functions`', '`-hook`', '`-hooks`', '`-form`', '`-forms`', '`-map`', '`-map-alist`', '`-mode-alist`', '`-program`', or '`-predicate`' is automatically considered risky. The variables '`font-lock-keywords`', '`font-lock-keywords`' followed by a digit, and '`font-lock-syntactic-keywords`' are also considered risky.

`risky-local-variable-p` *sym* [Function]
This function returns non-`nil` if *sym* is a risky variable, based on the above criteria.

`ignored-local-variables` [Variable]
This variable holds a list of variables that should not be given local values by files. Any value specified for one of these variables is completely ignored.

The '`Eval:`' "variable" is also a potential loophole, so Emacs normally asks for confirmation before handling it.

`enable-local-eval` [User Option]
This variable controls processing of '`Eval:`' in '`-*-`' lines or local variables lists in files being visited. A value of `t` means process them unconditionally; `nil` means ignore them; anything else means ask the user what to do for each file. The default value is `maybe`.

`safe-local-eval-forms` [User Option]
This variable holds a list of expressions that are safe to evaluate when found in the '`Eval:`' "variable" in a file local variables list.

If the expression is a function call and the function has a `safe-local-eval-function` property, the property value determines whether the expression is safe to evaluate. The property value can be a predicate to call to test the expression, a list of such predicates (it's safe if any predicate succeeds), or `t` (always safe provided the arguments are constant).

Text properties are also potential loopholes, since their values could include functions to call. So Emacs discards all text properties from string values specified for file-local variables.

12.13 Directory Local Variables

A directory can specify local variable values common to all files in that directory; Emacs uses these to create buffer-local bindings for those variables in buffers visiting any file in that directory. This is useful when the files in the directory belong to some *project* and therefore share the same local variables.

There are two different methods for specifying directory local variables: by putting them in a special file, or by defining a *project class* for that directory.

`dir-locals-file` [Constant]
: This constant is the name of the file where Emacs expects to find the directory-local variables. The name of the file is `.dir-locals.el`[3]. A file by that name in a directory causes Emacs to apply its settings to any file in that directory or any of its subdirectories (optionally, you can exclude subdirectories; see below). If some of the subdirectories have their own `.dir-locals.el` files, Emacs uses the settings from the deepest file it finds starting from the file's directory and moving up the directory tree. This constant is also used to derive the name of a second dir-locals file `.dir-locals-2.el`. If this second dir-locals file is present, then that is loaded in addition to `.dir-locals.el`. This is useful when `.dir-locals.el` is under version control in a shared repository and cannot be used for personal customizations. The file specifies local variables as a specially formatted list; see Section "Per-directory Local Variables" in *The GNU Emacs Manual*, for more details.

`hack-dir-local-variables` [Function]
: This function reads the `.dir-locals.el` file and stores the directory-local variables in `file-local-variables-alist` that is local to the buffer visiting any file in the directory, without applying them. It also stores the directory-local settings in `dir-locals-class-alist`, where it defines a special class for the directory in which `.dir-locals.el` file was found. This function works by calling `dir-locals-set-class-variables` and `dir-locals-set-directory-class`, described below.

`hack-dir-local-variables-non-file-buffer` [Function]
: This function looks for directory-local variables, and immediately applies them in the current buffer. It is intended to be called in the mode commands for non-file buffers, such as Dired buffers, to let them obey directory-local variable settings. For non-file buffers, Emacs looks for directory-local variables in `default-directory` and its parent directories.

`dir-locals-set-class-variables` *class variables* [Function]
: This function defines a set of variable settings for the named *class*, which is a symbol. You can later assign the class to one or more directories, and Emacs will apply those variable settings to all files in those directories. The list in *variables* can be of one of the two forms: `(`*`major-mode`*` . `*`alist`*`)` or `(`*`directory`*` . `*`list`*`)`. With the first form, if the file's buffer turns on a mode that is derived from *major-mode*, then all the variables in the associated *alist* are applied; *alist* should be of the form `(`*`name`*` . `*`value`*`)`. A special value `nil` for *major-mode* means the settings are applicable to

[3] The MS-DOS version of Emacs uses `_dir-locals.el` instead, due to limitations of the DOS filesystems.

any mode. In *alist*, you can use a special *name*: `subdirs`. If the associated value is `nil`, the alist is only applied to files in the relevant directory, not to those in any subdirectories.

With the second form of *variables*, if *directory* is the initial substring of the file's directory, then *list* is applied recursively by following the above rules; *list* should be of one of the two forms accepted by this function in *variables*.

`dir-locals-set-directory-class` *directory class* **&optional** *mtime* [Function]
This function assigns *class* to all the files in `directory` and its subdirectories. Thereafter, all the variable settings specified for *class* will be applied to any visited file in *directory* and its children. *class* must have been already defined by `dir-locals-set-class-variables`.

Emacs uses this function internally when it loads directory variables from a `.dir-locals.el` file. In that case, the optional argument *mtime* holds the file modification time (as returned by `file-attributes`). Emacs uses this time to check stored local variables are still valid. If you are assigning a class directly, not via a file, this argument should be `nil`.

`dir-locals-class-alist` [Variable]
This alist holds the class symbols and the associated variable settings. It is updated by `dir-locals-set-class-variables`.

`dir-locals-directory-cache` [Variable]
This alist holds directory names, their assigned class names, and modification times of the associated directory local variables file (if there is one). The function `dir-locals-set-directory-class` updates this list.

`enable-dir-local-variables` [Variable]
If `nil`, directory-local variables are ignored. This variable may be useful for modes that want to ignore directory-locals while still respecting file-local variables (see Section 12.12 [File Local Variables], page 213).

12.14 Connection Local Variables

Connection-local variables provide a general mechanism for different variable settings in buffers with a remote connection (see Section "Remote Files" in *The GNU Emacs Manual*). They are bound and set depending on the remote connection a buffer is dedicated to.

12.14.1 Connection Local Profiles

Emacs uses connection-local profiles to store the variable settings to apply to particular connections. You can then associate these with remote connections by defining the criteria when they should apply, using `connection-local-set-profiles`.

`connection-local-set-profile-variables` *profile variables* [Function]
This function defines a set of variable settings for the connection *profile*, which is a symbol. You can later assign the connection profile to one or more remote connections, and Emacs will apply those variable settings to all process buffers for those connections. The list in *variables* is an alist of the form `(name . value)`. Example:

```
(connection-local-set-profile-variables
  'remote-bash
  '((shell-file-name . "/bin/bash")
    (shell-command-switch . "-c")
    (shell-interactive-switch . "-i")
    (shell-login-switch . "-l")))

(connection-local-set-profile-variables
  'remote-ksh
  '((shell-file-name . "/bin/ksh")
    (shell-command-switch . "-c")
    (shell-interactive-switch . "-i")
    (shell-login-switch . "-l")))

(connection-local-set-profile-variables
  'remote-null-device
  '((null-device . "/dev/null")))
```

If you want to append variable settings to an existing profile, you could use the function `connection-local-get-profile-variables` in order to retrieve the existing settings, like

```
(connection-local-set-profile-variables
  'remote-bash
  (append
   (connection-local-get-profile-variables 'remote-bash)
   '((shell-command-dont-erase-buffer . t))))
```

`connection-local-profile-alist` [User Option]

This alist holds the connection profile symbols and the associated variable settings. It is updated by `connection-local-set-profile-variables`.

`connection-local-set-profiles` *criteria* **&rest** *profiles* [Function]

This function assigns *profiles*, which are symbols, to all remote connections identified by *criteria*. *criteria* is a plist identifying a connection and the application using this connection. Property names might be `:application`, `:protocol`, `:user` and `:machine`. The property value of `:application` is a symbol, all other property values are strings. All properties are optional; if *criteria* is `nil`, it always applies. Example:

```
(connection-local-set-profiles
  '(:application tramp :protocol "ssh" :machine "localhost")
  'remote-bash 'remote-null-device)

(connection-local-set-profiles
  '(:application tramp :protocol "sudo"
    :user "root" :machine "localhost")
  'remote-ksh 'remote-null-device)
```

If *criteria* is `nil`, it applies for all remote connections. Therefore, the example above would be equivalent to

```
(connection-local-set-profiles
  '(:application tramp :protocol "ssh" :machine "localhost")
  'remote-bash)

(connection-local-set-profiles
  '(:application tramp :protocol "sudo"
    :user "root" :machine "localhost")
  'remote-ksh)

(connection-local-set-profiles
  nil 'remote-null-device)
```

Any connection profile of *profiles* must have been already defined by `connection-local-set-profile-variables`.

`connection-local-criteria-alist` [User Option]
This alist contains connection criteria and their assigned profile names. The function `connection-local-set-profiles` updates this list.

12.14.2 Applying Connection Local Variables

When writing connection-aware code, you'll need to collect, and possibly apply, any connection-local variables. There are several ways to do this, as described below.

`hack-connection-local-variables` *criteria* [Function]
This function collects applicable connection-local variables associated with *criteria* in `connection-local-variables-alist`, without applying them. Example:

```
(hack-connection-local-variables
  '(:application tramp :protocol "ssh" :machine "localhost"))

connection-local-variables-alist
     ⇒ ((null-device . "/dev/null")
        (shell-login-switch . "-l")
        (shell-interactive-switch . "-i")
        (shell-command-switch . "-c")
        (shell-file-name . "/bin/bash"))
```

`hack-connection-local-variables-apply` *criteria* [Function]
This function looks for connection-local variables according to *criteria*, and immediately applies them in the current buffer.

`with-connection-local-application-variables` *application* **&rest** *body* [Macro]
Apply all connection-local variables for `application`, which are specified by `default-directory`.

After that, *body* is executed, and the connection-local variables are unwound. Example:

```
(connection-local-set-profile-variables
  'my-remote-perl
  '((perl-command-name . "/usr/local/bin/perl5")
    (perl-command-switch . "-e %s")))

(connection-local-set-profiles
  '(:application my-app :protocol "ssh" :machine "remotehost")
  'my-remote-perl)

(let ((default-directory "/ssh:remotehost:/working/dir/"))
  (with-connection-local-application-variables 'my-app
    do something useful))
```

`connection-local-default-application` [Variable]

The default application, a symbol, to be applied in `with-connection-local-variables`. It defaults to `tramp`, but you can let-bind it to change the application temporarily (see Section 12.3 [Local Variables], page 190).

This variable must not be changed globally.

`with-connection-local-variables` **&rest** *body* [Macro]

This is equivalent to `with-connection-local-application-variables`, but uses `connection-local-default-application` for the application.

`setq-connection-local` [*symbol form*]... [Macro]

This macro sets each *symbol* connection-locally to the result of evaluating the corresponding *form*, using the connection-local profile specified in `connection-local-profile-name-for-setq`; if the profile name is `nil`, this macro will just set the variables normally, as with `setq` (see Section 12.8 [Setting Variables], page 198).

For example, you can use this macro in combination with `with-connection-local-variables` or `with-connection-local-application-variables` to lazily initialize connection-local settings:

```
(defvar my-app-variable nil)

(connection-local-set-profile-variables
 'my-app-connection-default-profile
 '((my-app-variable . nil)))

(connection-local-set-profiles
 '(:application my-app)
 'my-app-connection-default-profile)

(defun my-app-get-variable ()
  (with-connection-local-application-variables 'my-app
    (or my-app-variable
        (setq-connection-local my-app-variable
                               do something useful))))
```

`connection-local-profile-name-for-setq` [Variable]
: The connection-local profile name, a symbol, to use when setting variables via `setq-connection-local`. This is let-bound in the body of `with-connection-local-variables`, but you can also let-bind it yourself if you'd like to set variables on a different profile.

  This variable must not be changed globally.

`enable-connection-local-variables` [Variable]
: If `nil`, connection-local variables are ignored. This variable shall be changed temporarily only in special modes.

12.15 Variable Aliases

It is sometimes useful to make two variables synonyms, so that both variables always have the same value, and changing either one also changes the other. Whenever you change the name of a variable—either because you realize its old name was not well chosen, or because its meaning has partly changed—it can be useful to keep the old name as an *alias* of the new one for compatibility. You can do this with `defvaralias`.

`defvaralias` *new-alias base-variable* **&optional** *docstring* [Function]
: This function defines the symbol *new-alias* as a variable alias for symbol *base-variable*. This means that retrieving the value of *new-alias* returns the value of *base-variable*, and changing the value of *new-alias* changes the value of *base-variable*. The two aliased variable names always share the same value and the same bindings.

  If the *docstring* argument is non-`nil`, it specifies the documentation for *new-alias*; otherwise, the alias gets the same documentation as *base-variable* has, if any, unless *base-variable* is itself an alias, in which case *new-alias* gets the documentation of the variable at the end of the chain of aliases.

  This function returns *base-variable*.

Variable aliases are convenient for replacing an old name for a variable with a new name. `make-obsolete-variable` declares that the old name is obsolete and therefore that it may be removed at some stage in the future.

`make-obsolete-variable` *obsolete-name current-name when* [Function]
&optional *access-type*
: This function makes the byte compiler warn that the variable *obsolete-name* is obsolete. If *current-name* is a symbol, it is the variable's new name; then the warning message says to use *current-name* instead of *obsolete-name*. If *current-name* is a string, this is the message and there is no replacement variable. *when* should be a string indicating when the variable was first made obsolete (usually a version number string).

  The optional argument *access-type*, if non-`nil`, should specify the kind of access that will trigger obsolescence warnings; it can be either `get` or `set`.

You can make two variables synonyms and declare one obsolete at the same time using the macro `define-obsolete-variable-alias`.

`define-obsolete-variable-alias` *obsolete-name current-name when* **&optional** *docstring* [Macro]

This macro marks the variable *obsolete-name* as obsolete and also makes it an alias for the variable *current-name*. It is equivalent to the following:

```
(defvaralias obsolete-name current-name docstring)
(make-obsolete-variable obsolete-name current-name when)
```

This macro evaluates all its parameters, and both *obsolete-name* and *current-name* should be symbols, so a typical usage would look like:

```
(define-obsolete-variable-alias 'foo-thing 'bar-thing "27.1")
```

`indirect-variable` *variable* [Function]

This function returns the variable at the end of the chain of aliases of *variable*. If *variable* is not a symbol, or if *variable* is not defined as an alias, the function returns *variable*.

This function signals a `cyclic-variable-indirection` error if there is a loop in the chain of symbols.

```
(defvaralias 'foo 'bar)
(indirect-variable 'foo)
     ⇒ bar
(indirect-variable 'bar)
     ⇒ bar
(setq bar 2)
bar
     ⇒ 2
foo
     ⇒ 2
(setq foo 0)
bar
     ⇒ 0
foo
     ⇒ 0
```

12.16 Variables with Restricted Values

Ordinary Lisp variables can be assigned any value that is a valid Lisp object. However, certain Lisp variables are not defined in Lisp, but in C. Most of these variables are defined in the C code using `DEFVAR_LISP`. Like variables defined in Lisp, these can take on any value. However, some variables are defined using `DEFVAR_INT` or `DEFVAR_BOOL`. See [Writing Emacs Primitives], page 1364, in particular the description of functions of the type `syms_of_`*`filename`*, for a brief discussion of the C implementation.

Variables of type `DEFVAR_BOOL` can only take on the values `nil` or `t`. Attempting to assign them any other value will set them to `t`:

```
(let ((display-hourglass 5))
  display-hourglass)
     ⇒ t
```

`byte-boolean-vars` [Variable]
: This variable holds a list of all variables of type `DEFVAR_BOOL`.

Variables of type `DEFVAR_INT` can take on only integer values. Attempting to assign them any other value will result in an error:

```
(setq undo-limit 1000.0)
error  Wrong type argument: integerp, 1000.0
```

12.17 Generalized Variables

A *generalized variable* or *place form* is one of the many places in Lisp memory where values can be stored using the `setf` macro (see Section 12.17.1 [Setting Generalized Variables], page 224). The simplest place form is a regular Lisp variable. But the CARs and CDRs of lists, elements of arrays, properties of symbols, and many other locations are also places where Lisp values get stored.

Generalized variables are analogous to lvalues in the C language, where '`x = a[i]`' gets an element from an array and '`a[i] = x`' stores an element using the same notation. Just as certain forms like `a[i]` can be lvalues in C, there is a set of forms that can be generalized variables in Lisp.

12.17.1 The `setf` Macro

The `setf` macro is the most basic way to operate on generalized variables. The `setf` form is like `setq`, except that it accepts arbitrary place forms in the first (left) argument of each pair rather than just symbols. For example, `(setf (car a) b)` sets the car of `a` to `b`, doing the same operation as `(setcar a b)`, but without you having to use two separate functions for setting and accessing this type of place.

`setf` [*place form*]... [Macro]
: This macro evaluates *form* and stores its value in *place*, which must be a valid generalized variable form. If there are several *place* and *form* pairs, the assignments are done sequentially just as with `setq`. `setf` returns the value of the last *form*.

The following Lisp forms are the forms in Emacs that will work as generalized variables, and so may appear in the *place* argument of `setf`:

- A symbol. In other words, `(setf x y)` is exactly equivalent to `(setq x y)`, and `setq` itself is strictly speaking redundant given that `setf` exists. Most programmers will continue to prefer `setq` for setting simple variables, though, for stylistic and historical reasons. The macro `(setf x y)` actually expands to `(setq x y)`, so there is no performance penalty for using it in compiled code.
- A call to any of the following standard Lisp functions:

```
aref      cddr      symbol-function
car       elt       symbol-plist
caar      get       symbol-value
cadr      gethash
cdr       nth
cdar      nthcdr
```

- A call to any of the following Emacs-specific functions:

```
alist-get                    overlay-start
```

```
default-value                overlay-get
face-background              process-buffer
face-font                    process-filter
face-foreground              process-get
face-stipple                 process-sentinel
face-underline-p             terminal-parameter
file-modes                   window-buffer
frame-parameter              window-dedicated-p
frame-parameters             window-display-table
get-register                 window-hscroll
getenv                       window-parameter
keymap-parent                window-point
match-data                   window-start
overlay-end
```

- A call of the form `(substring` *subplace* *n* `[`*m*`])`, where *subplace* is itself a valid generalized variable whose current value is a string, and where the value stored is also a string. The new string is spliced into the specified part of the destination string. For example:

```
(setq a (list "hello" "world"))
     ⇒ ("hello" "world")
(cadr a)
     ⇒ "world"
(substring (cadr a) 2 4)
     ⇒ "rl"
(setf (substring (cadr a) 2 4) "o")
     ⇒ "o"
(cadr a)
     ⇒ "wood"
a
     ⇒ ("hello" "wood")
```

- The `if` and `cond` conditionals will work as generalized variables. For instance, this will set either the `foo` or the `bar` variable to `zot`:

```
(setf (if (zerop (random 2))
  foo
bar)
      'zot)
```

`setf` signals an error if you pass a *place* form that it does not know how to handle.

Note that for `nthcdr`, the list argument of the function must itself be a valid *place* form. For example, `(setf (nthcdr 0 foo) 7)` will set `foo` itself to 7.

The macros `push` (see Section 5.5 [List Variables], page 85) and `pop` (see Section 5.3 [List Elements], page 79) can manipulate generalized variables, not just lists. `(pop` *place*`)` removes and returns the first element of the list stored in *place*. It is analogous to `(prog1 (car` *place*`) (setf` *place* `(cdr` *place*`)))`, except that it takes care to evaluate all subforms only once. `(push` *x* *place*`)` inserts *x* at the front of the list stored in *place*. It is analogous to `(setf` *place* `(cons` *x* *place*`))`, except for evaluation of the subforms. Note that `push` and `pop` on an `nthcdr` place can be used to insert or delete at any position in a list.

The `cl-lib` library defines various extensions for generalized variables, including additional `setf` places. See Section "Generalized Variables" in *Common Lisp Extensions*.

12.17.2 Defining new `setf` forms

This section describes how to define new forms that `setf` can operate on.

`gv-define-simple-setter` *name setter* **&optional** *fix-return* [Macro]

This macro enables you to easily define `setf` methods for simple cases. *name* is the name of a function, macro, or special form. You can use this macro whenever *name* has a directly corresponding *setter* function that updates it, e.g., `(gv-define-simple-setter car setcar)`.

This macro translates a call of the form

```
(setf (name args...) value)
```

into

```
(setter args... value)
```

Such a `setf` call is documented to return *value*. This is no problem with, e.g., `car` and `setcar`, because `setcar` returns the value that it set. If your *setter* function does not return *value*, use a non-`nil` value for the *fix-return* argument of `gv-define-simple-setter`. This expands into something equivalent to

```
(let ((temp value))
  (setter args... temp)
  temp)
```

so ensuring that it returns the correct result.

`gv-define-setter` *name arglist* **&rest** *body* [Macro]

This macro allows for more complex `setf` expansions than the previous form. You may need to use this form, for example, if there is no simple setter function to call, or if there is one but it requires different arguments to the place form.

This macro expands the form `(setf (name args...) value)` by first binding the `setf` argument forms `(value args...)` according to *arglist*, and then executing *body*. *body* should return a Lisp form that does the assignment, and finally returns the value that was set. An example of using this macro is:

```
(gv-define-setter caar (val x) `(setcar (car ,x) ,val))
```

`gv-define-expander` *name handler* [Macro]

For more control over the expansion, the `gv-define-expander` macro can be used. For instance, a settable `substring` could be implemented this way:

```
(gv-define-expander substring
  (lambda (do place from &optional to)
    (gv-letplace (getter setter) place
      (macroexp-let2* (from to)
        (funcall do `(substring ,getter ,from ,to)
                 (lambda (v)
                   (macroexp-let2* (v)
                     `(progn
                        ,(funcall setter `(cl--set-substring
                                           ,getter ,from ,to ,v))
                        ,v))))))))
```

`gv-letplace` (*getter setter*) *place* **&rest** *body* [Macro]

The macro `gv-letplace` can be useful in defining macros that perform similarly to `setf`; for example, the `incf` macro of Common Lisp could be implemented this way:

```
(defmacro incf (place &optional n)
  (gv-letplace (getter setter) place
    (macroexp-let2* ((v (or n 1)))
      (funcall setter `(+ ,v ,getter)))))
```

getter will be bound to a copyable expression that returns the value of *place*. *setter* will be bound to a function that takes an expression *v* and returns a new expression that sets *place* to *v*. *body* should return a Emacs Lisp expression manipulating *place* via *getter* and *setter*.

Consult the source file `gv.el` for more details.

`make-obsolete-generalized-variable` *obsolete-name current-name when* [Function]

This function makes the byte compiler warn that the generalized variable *obsolete-name* is obsolete. If *current-name* is a symbol, then the warning message says to use *current-name* instead of *obsolete-name*. If *current-name* is a string, this is the message. *when* should be a string indicating when the variable was first made obsolete (usually a version number string).

Common Lisp note: Common Lisp defines another way to specify the `setf` behavior of a function, namely `setf` functions, whose names are lists `(setf` *`name`*`)` rather than symbols. For example, `(defun (setf foo) ...)` defines the function that is used when `setf` is applied to `foo`. Emacs does not support this. It is a compile-time error to use `setf` on a form that has not already had an appropriate expansion defined. In Common Lisp, this is not an error since the function `(setf` *`func`*`)` might be defined later.

12.18 Multisession Variables

When you set a variable to a value and then close Emacs and restart it, that value won't be automatically restored. Users usually set normal variables in their startup files, or use Customize (see Chapter 15 [Customization], page 278) to set user options permanently, and various packages have various files where they store the data (e.g., Gnus stores this in `.newsrc.eld` and the URL library stores cookies in `~/.emacs.d/url/cookies`).

For things in between these two extremes (i.e., configuration which goes in the startup file, and massive application state that goes into separate files), Emacs provides a facility to replicate data between sessions called *multisession variables*. (This facility may not be available on all systems.) To give you an idea of how these are meant to be used, here's a small example:

```
(define-multisession-variable foo 0)
(defun my-adder (num)
  (interactive "nAdd number: ")
  (setf (multisession-value foo)
        (+ (multisession-value foo) num))
  (message "The new number is: %s" (multisession-value foo)))
```

This defines the variable `foo` and binds it to a special multisession object which is initialized with the value '`0`' (if the variable doesn't already exist from a previous session). The `my-adder` command queries the user for a number, adds this to the old (possibly saved value), and then saves the new value.

This facility isn't meant to be used for huge data structures, but should be performant for most values.

`define-multisession-variable` *name initial-value* **&optional** *doc* **&rest** *args* [Macro]

This macro defines *name* as a multisession variable, and gives it the *initial-value* if this variable hasn't been assigned a value earlier. *doc* is the doc string, and several keyword arguments can be used in *args*:

`:package` *package-symbol*

This keyword says that a multisession variable belongs to the package specified by *package-symbol*. The combination of *package-symbol* and *name* has to be unique. If *package-symbol* isn't given, this will default to the first "segment" of the *name* symbol's name, which is the part of its name up to and excluding the first '`-`'. For instance, if *name* is `foo` and *package-symbol* isn't given, *package-symbol* will default to `foo`.

`:synchronized` *bool*

Multisession variables can be *synchronized* if *bool* is non-`nil`. This means that if there're two concurrent Emacs instances running, and the other Emacs changes the multisession variable `foo`, the current Emacs instance will retrieve that modified data when accessing the value. If *synchronized* is `nil` or missing, this won't happen, and the values in all Emacs sessions using the variable will be independent of each other.

`:storage` *storage*

Use the specified *storage* method. This can be either `sqlite` (in Emacs compiled with SQLite support) or `files`. If not given, this defaults to the value of the `multisession-storage` variable, described below.

`multisession-value` *variable* [Function]

This function returns the current value of *variable*. If this variable hasn't been accessed before in this Emacs session, or if it's changed externally, it will be read in from external storage. If not, the current value in this session is returned as is. It is an error to call this function for a *variable* that is not a multisession variable.

Values retrieved via `multisession-value` may or may not be `eq` to each other, but they will always be `equal`.

This is a generalized variable (see Section 12.17 [Generalized Variables], page 224), so the way to update such a variable is to say, for instance:

```
(setf (multisession-value foo-bar) 'zot)
```

Only Emacs Lisp values that have a readable print syntax (see Section 2.1 [Printed Representation], page 8) can be saved this way.

If the multisession variable is synchronized, setting it may update the value first. For instance:

```
(cl-incf (multisession-value foo-bar))
```

This first checks whether the value has changed in a different Emacs instance, retrieves that value, and then adds 1 to that value and stores it. But note that this is done without locking, so if many instances are updating the value at the same time, it's unpredictable which instance "wins".

`multisession-delete` *object* [Function]
This function deletes *object* and its value from its persistent storage.

`make-multisession` [Function]
You can also make persistent values that aren't tied to a specific variable, but are tied to an explicit package and key.

```
(setq foo (make-multisession :package "mail"
                             :key "friends"))
(setf (multisession-value foo) 'everybody)
```

This supports the same keywords as `define-multisession-variable`, but also supports a `:initial-value` keyword, which specifies the default value.

`multisession-storage` [User Option]
This variable controls how the multisession variables are stored. It value defaults to `files`, which means that the values are stored in a one-file-per-variable structure inside the directory specified by `multisession-directory`. If this value is `sqlite` instead, the values are stored in an SQLite database; this is only available if Emacs was built with SQLite support.

`multisession-directory` [User Option]
The multisession variables are stored under this directory, which defaults to `multisession/` subdirectory of the `user-emacs-directory`, which is typically `~/.emacs.d/multisession/`.

`list-multisession-values` [Command]
This command pops up a buffer listing all the multisession variables, and enters a special mode `multisession-edit-mode` which allows you to delete them and edit their values.

13 Functions

A Lisp program is composed mainly of Lisp functions. This chapter explains what functions are, how they accept arguments, and how to define them.

13.1 What Is a Function?

In a general sense, a function is a rule for carrying out a computation given input values called *arguments*. The result of the computation is called the *value* or *return value* of the function. The computation can also have side effects, such as lasting changes in the values of variables or the contents of data structures (see [Definition of side effect], page 145). A *pure function* is a function which, in addition to having no side effects, always returns the same value for the same combination of arguments, regardless of external factors such as machine type or system state.

In most computer languages, every function has a name. But in Lisp, a function in the strictest sense has no name: it is an object which can *optionally* be associated with a symbol (e.g., `car`) that serves as the function name. See Section 13.3 [Function Names], page 236. When a function has been given a name, we usually also refer to that symbol as a "function" (e.g., we refer to "the function `car`"). In this manual, the distinction between a function name and the function object itself is usually unimportant, but we will take note wherever it is relevant.

Certain function-like objects, called *special forms* and *macros*, also accept arguments to carry out computations. However, as explained below, these are not considered functions in Emacs Lisp.

Here are important terms for functions and function-like objects:

lambda expression
: A function (in the strict sense, i.e., a function object) which is written in Lisp. These are described in the following section.

primitive
: A function which is callable from Lisp but is actually written in C. Primitives are also called *built-in functions*, or *subrs*. Examples include functions like `car` and `append`. In addition, all special forms (see below) are also considered primitives.

  Usually, a function is implemented as a primitive because it is a fundamental part of Lisp (e.g., `car`), or because it provides a low-level interface to operating system services, or because it needs to run fast. Unlike functions defined in Lisp, primitives can be modified or added only by changing the C sources and recompiling Emacs. See Section E.7 [Writing Emacs Primitives], page 1361.

special form
: A primitive that is like a function but does not evaluate all of its arguments in the usual way. It may evaluate only some of the arguments, or may evaluate them in an unusual order, or several times. Examples include `if`, `and`, and `while`. See Section 10.1.7 [Special Forms], page 149.

macro
: A construct defined in Lisp, which differs from a function in that it translates a Lisp expression into another expression which is to be evaluated instead of the

original expression. Macros enable Lisp programmers to do the sorts of things that special forms can do. See Chapter 14 [Macros], page 269.

command An object which can be invoked via the `command-execute` primitive, usually due to the user typing in a key sequence *bound* to that command. See Section 22.3 [Interactive Call], page 433. A command is usually a function; if the function is written in Lisp, it is made into a command by an `interactive` form in the function definition (see Section 22.2 [Defining Commands], page 425). Commands that are functions can also be called from Lisp expressions, just like other functions.

Keyboard macros (strings and vectors) are commands also, even though they are not functions. See Section 22.16 [Keyboard Macros], page 479. We say that a symbol is a command if its function cell contains a command (see Section 9.1 [Symbol Components], page 133); such a *named command* can be invoked with `M-x`.

closure A function object that is much like a lambda expression, except that it also encloses an environment of lexical variable bindings. See Section 13.10 [Closures], page 250.

byte-code function
A function that has been compiled by the byte compiler. See Section 2.4.16 [Byte-Code Type], page 24.

autoload object
A place-holder for a real function. If the autoload object is called, Emacs loads the file containing the definition of the real function, and then calls the real function. See Section 16.5 [Autoload], page 305.

You can use the function `functionp` to test if an object is a function:

`functionp` *object* [Function]
This function returns `t` if *object* is any kind of function, i.e., can be passed to `funcall`. Note that `functionp` returns `t` for symbols that are function names, and returns `nil` for symbols that are macros or special forms.

If *object* is not a function, this function ordinarily returns `nil`. However, the representation of function objects is complicated, and for efficiency reasons in rare cases this function can return `t` even when *object* is not a function.

It is also possible to find out how many arguments an arbitrary function expects:

`func-arity` *function* [Function]
This function provides information about the argument list of the specified *function*. The returned value is a cons cell of the form `(min . max)`, where *min* is the minimum number of arguments, and *max* is either the maximum number of arguments, or the symbol `many` for functions with `&rest` arguments, or the symbol `unevalled` if *function* is a special form.

Note that this function might return inaccurate results in some situations, such as the following:

– Functions defined using `apply-partially` (see Section 13.5 [Calling Functions], page 239).

- Functions that are advised using `advice-add` (see Section 13.12.2 [Advising Named Functions], page 255).
- Functions that determine the argument list dynamically, as part of their code.

Unlike `functionp`, the next three functions do *not* treat a symbol as its function definition.

`subrp` *object* [Function]
This function returns `t` if *object* is a built-in function (i.e., a Lisp primitive).

```
(subrp 'message)            ; message is a symbol,
     ⇒ nil                  ;    not a subr object.
(subrp (symbol-function 'message))
     ⇒ t
```

`byte-code-function-p` *object* [Function]
This function returns `t` if *object* is a byte-code function. For example:

```
(byte-code-function-p (symbol-function 'next-line))
     ⇒ t
```

`compiled-function-p` *object* [Function]
This function returns `t` if *object* is a function object that is not in the form of ELisp source code but something like machine code or byte code instead. More specifically it returns `t` if the function is built-in (a.k.a. "primitive", see Section 13.1 [What Is a Function], page 230), or byte-compiled (see Chapter 17 [Byte Compilation], page 317), or natively-compiled (see Chapter 18 [Native Compilation], page 328), or a function loaded from a dynamic module (see Section 16.11 [Dynamic Modules], page 315).

`subr-arity` *subr* [Function]
This works like `func-arity`, but only for built-in functions and without symbol indirection. It signals an error for non-built-in functions. We recommend to use `func-arity` instead.

13.2 Lambda Expressions

A lambda expression is a function object written in Lisp. Here is an example:

```
(lambda (x)
  "Return the hyperbolic cosine of X."
  (* 0.5 (+ (exp x) (exp (- x)))))
```

In Emacs Lisp, such a list is a valid expression which evaluates to a function object.

A lambda expression, by itself, has no name; it is an *anonymous function*. Although lambda expressions can be used this way (see Section 13.7 [Anonymous Functions], page 243), they are more commonly associated with symbols to make *named functions* (see Section 13.3 [Function Names], page 236). Before going into these details, the following subsections describe the components of a lambda expression and what they do.

13.2.1 Components of a Lambda Expression

A lambda expression is a list that looks like this:

```
(lambda (arg-variables...)
```

```
[documentation-string]
[interactive-declaration]
body-forms...)
```

The first element of a lambda expression is always the symbol `lambda`. This indicates that the list represents a function. The reason functions are defined to start with `lambda` is so that other lists, intended for other uses, will not accidentally be valid as functions.

The second element is a list of symbols—the argument variable names (see Section 13.2.3 [Argument List], page 234). This is called the *lambda list*. When a Lisp function is called, the argument values are matched up against the variables in the lambda list, which are given local bindings with the values provided. See Section 12.3 [Local Variables], page 190.

The documentation string is a Lisp string object placed within the function definition to describe the function for the Emacs help facilities. See Section 13.2.4 [Function Documentation], page 235.

The interactive declaration is a list of the form `(interactive` *`code-string`*`)`. This declares how to provide arguments if the function is used interactively. Functions with this declaration are called *commands*; they can be called using `M-x` or bound to a key. Functions not intended to be called in this way should not have interactive declarations. See Section 22.2 [Defining Commands], page 425, for how to write an interactive declaration.

The rest of the elements are the *body* of the function: the Lisp code to do the work of the function (or, as a Lisp programmer would say, "a list of Lisp forms to evaluate"). The value returned by the function is the value returned by the last element of the body.

13.2.2 A Simple Lambda Expression Example

Consider the following example:

```
(lambda (a b c) (+ a b c))
```

We can call this function by passing it to `funcall`, like this:

```
(funcall (lambda (a b c) (+ a b c))
         1 2 3)
```

This call evaluates the body of the lambda expression with the variable `a` bound to 1, `b` bound to 2, and `c` bound to 3. Evaluation of the body adds these three numbers, producing the result 6; therefore, this call to the function returns the value 6.

Note that the arguments can be the results of other function calls, as in this example:

```
(funcall (lambda (a b c) (+ a b c))
         1 (* 2 3) (- 5 4))
```

This evaluates the arguments `1`, `(* 2 3)`, and `(- 5 4)` from left to right. Then it applies the lambda expression to the argument values 1, 6 and 1 to produce the value 8.

As these examples show, you can use a form with a lambda expression as its CAR to make local variables and give them values. In the old days of Lisp, this technique was the only way to bind and initialize local variables. But nowadays, it is clearer to use the special form `let` for this purpose (see Section 12.3 [Local Variables], page 190). Lambda expressions are mainly used as anonymous functions for passing as arguments to other functions (see Section 13.7 [Anonymous Functions], page 243), or stored as symbol function definitions to produce named functions (see Section 13.3 [Function Names], page 236).

13.2.3 Features of Argument Lists

Our simple sample function, `(lambda (a b c) (+ a b c))`, specifies three argument variables, so it must be called with three arguments: if you try to call it with only two arguments or four arguments, you get a `wrong-number-of-arguments` error (see Section 11.7.3 [Errors], page 179).

It is often convenient to write a function that allows certain arguments to be omitted. For example, the function `substring` accepts three arguments—a string, the start index and the end index—but the third argument defaults to the *length* of the string if you omit it. It is also convenient for certain functions to accept an indefinite number of arguments, as the functions `list` and `+` do.

To specify optional arguments that may be omitted when a function is called, simply include the keyword `&optional` before the optional arguments. To specify a list of zero or more extra arguments, include the keyword `&rest` before one final argument.

Thus, the complete syntax for an argument list is as follows:

```
(required-vars...
 [&optional [optional-vars...]]
 [&rest rest-var])
```

The square brackets indicate that the `&optional` and `&rest` clauses, and the variables that follow them, are optional.

A call to the function requires one actual argument for each of the *required-vars*. There may be actual arguments for zero or more of the *optional-vars*, and there cannot be any actual arguments beyond that unless the lambda list uses `&rest`. In that case, there may be any number of extra actual arguments.

If actual arguments for the optional and rest variables are omitted, then they always default to `nil`. There is no way for the function to distinguish between an explicit argument of `nil` and an omitted argument. However, the body of the function is free to consider `nil` an abbreviation for some other meaningful value. This is what `substring` does; `nil` as the third argument to `substring` means to use the length of the string supplied.

> **Common Lisp note:** Common Lisp allows the function to specify what default value to use when an optional argument is omitted; Emacs Lisp always uses `nil`. Emacs Lisp does not support `supplied-p` variables that tell you whether an argument was explicitly passed.

For example, an argument list that looks like this:

```
(a b &optional c d &rest e)
```

binds `a` and `b` to the first two actual arguments, which are required. If one or two more arguments are provided, `c` and `d` are bound to them respectively; any arguments after the first four are collected into a list and `e` is bound to that list. Thus, if there are only two arguments, `c`, `d` and `e` are `nil`; if two or three arguments, `d` and `e` are `nil`; if four arguments or fewer, `e` is `nil`. Note that exactly five arguments with an explicit `nil` argument provided for `e` will cause that `nil` argument to be passed as a list with one element, `(nil)`, as with any other single value for `e`.

There is no way to have required arguments following optional ones—it would not make sense. To see why this must be so, suppose that `c` in the example were optional and `d` were required. Suppose three actual arguments are given; which variable would the third

argument be for? Would it be used for the *c*, or for *d*? One can argue for both possibilities. Similarly, it makes no sense to have any more arguments (either required or optional) after a `&rest` argument.

Here are some examples of argument lists and proper calls:

```
(funcall (lambda (n) (1+ n))          ; One required:
         1)                           ; requires exactly one argument.
     ⇒ 2
(funcall (lambda (n &optional n1)     ; One required and one optional:
           (if n1 (+ n n1) (1+ n)))   ; 1 or 2 arguments.
         1 2)
     ⇒ 3
(funcall (lambda (n &rest ns)         ; One required and one rest:
           (+ n (apply '+ ns)))       ; 1 or more arguments.
         1 2 3 4 5)
     ⇒ 15
```

13.2.4 Documentation Strings of Functions

A lambda expression may optionally have a *documentation string* just after the lambda list. This string does not affect execution of the function; it is a kind of comment, but a systematized comment which actually appears inside the Lisp world and can be used by the Emacs help facilities. See Chapter 25 [Documentation], page 597, for how the documentation string is accessed.

It is a good idea to provide documentation strings for all the functions in your program, even those that are called only from within your program. Documentation strings are like comments, except that they are easier to access.

The first line of the documentation string should stand on its own, because `apropos` displays just this first line. It should consist of one or two complete sentences that summarize the function's purpose.

The start of the documentation string is usually indented in the source file, but since these spaces come before the starting double-quote, they are not part of the string. Some people make a practice of indenting any additional lines of the string so that the text lines up in the program source. *That is a mistake.* The indentation of the following lines is inside the string; what looks nice in the source code will look ugly when displayed by the help commands.

You may wonder how the documentation string could be optional, since there are required components of the function that follow it (the body). Since evaluation of a string returns that string, without any side effects, it has no effect if it is not the last form in the body. Thus, in practice, there is no confusion between the first form of the body and the documentation string; if the only body form is a string then it serves both as the return value and as the documentation.

The last line of the documentation string can specify calling conventions different from the actual function arguments. Write text like this:

```
\(fn arglist)
```

following a blank line, at the beginning of the line, with no newline following it inside the documentation string. (The '\' is used to avoid confusing the Emacs motion commands.)

The calling convention specified in this way appears in help messages in place of the one derived from the actual arguments of the function.

This feature is particularly useful for macro definitions, since the arguments written in a macro definition often do not correspond to the way users think of the parts of the macro call.

Do not use this feature if you want to deprecate the calling convention and favor the one you advertise by the above specification. Instead, use the `advertised-calling-convention` declaration (see Section 13.15 [Declare Form], page 263) or `set-advertised-calling-convention` (see Section 13.13 [Obsolete Functions], page 260), because these two will cause the byte compiler emit a warning message when it compiles Lisp programs which use the deprecated calling convention.

Documentation strings are usually static, but occasionally it can be necessary to generate them dynamically. In some cases you can do so by writing a macro which generates at compile time the code of the function, including the desired documentation string. But you can also generate the docstring dynamically by writing `(:documentation` *`form`*`)` instead of the documentation string. This will evaluate *form* at run-time when the function is defined and use it as the documentation string[1]. You can also compute the documentation string on the fly when it is requested, by setting the `function-documentation` property of the function's symbol to a Lisp form that evaluates to a string.

For example:

```
(defun adder (x)
  (lambda (y)
    (:documentation (format "Add %S to the argument Y." x))
    (+ x y)))
(defalias 'adder5 (adder 5))
(documentation 'adder5)
    ⇒ "Add 5 to the argument Y."

(put 'adder5 'function-documentation
     '(concat (documentation (symbol-function 'adder5) 'raw)
              "  Consulted at " (format-time-string "%H:%M:%S")))
(documentation 'adder5)
    ⇒ "Add 5 to the argument Y.  Consulted at 15:52:13"
(documentation 'adder5)
    ⇒ "Add 5 to the argument Y.  Consulted at 15:52:18"
```

13.3 Naming a Function

A symbol can serve as the name of a function. This happens when the symbol's *function cell* (see Section 9.1 [Symbol Components], page 133) contains a function object (e.g., a lambda expression). Then the symbol itself becomes a valid, callable function, equivalent to the function object in its function cell.

The contents of the function cell are also called the symbol's *function definition*. The procedure of using a symbol's function definition in place of the symbol is called *symbol*

[1] This only works in code using `lexical-binding`.

function indirection; see Section 10.1.4 [Function Indirection], page 147. If you have not given a symbol a function definition, its function cell is said to be *void*, and it cannot be used as a function.

In practice, nearly all functions have names, and are referred to by their names. You can create a named Lisp function by defining a lambda expression and putting it in a function cell (see Section 13.9 [Function Cells], page 248). However, it is more common to use the `defun` macro, described in the next section.

We give functions names because it is convenient to refer to them by their names in Lisp expressions. Also, a named Lisp function can easily refer to itself—it can be recursive. Furthermore, primitives can only be referred to textually by their names, since primitive function objects (see Section 2.4.15 [Primitive Function Type], page 24) have no read syntax.

A function need not have a unique name. A given function object *usually* appears in the function cell of only one symbol, but this is just a convention. It is easy to store it in several symbols using `fset`; then each of the symbols is a valid name for the same function.

Note that a symbol used as a function name may also be used as a variable; these two uses of a symbol are independent and do not conflict. (This is not the case in some dialects of Lisp, like Scheme.)

By convention, if a function's symbol consists of two names separated by '`--`', the function is intended for internal use and the first part names the file defining the function. For example, a function named `vc-git--rev-parse` is an internal function defined in `vc-git.el`. Internal-use functions written in C have names ending in '`-internal`', e.g., `bury-buffer-internal`. Emacs code contributed before 2018 may follow other internal-use naming conventions, which are being phased out.

13.4 Defining Functions

We usually give a name to a function when it is first created. This is called *defining a function*, and we usually do it with the `defun` macro. This section also describes other ways to define a function.

`defun` *name args* [*doc*] [*declare*] [*interactive*] *body*... [Macro]

`defun` is the usual way to define new Lisp functions. It defines the symbol *name* as a function with argument list *args* (see Section 13.2.3 [Argument List], page 234) and body forms given by *body*. Neither *name* nor *args* should be quoted.

doc, if present, should be a string specifying the function's documentation string (see Section 13.2.4 [Function Documentation], page 235). *declare*, if present, should be a `declare` form specifying function metadata (see Section 13.15 [Declare Form], page 263). *interactive*, if present, should be an `interactive` form specifying how the function is to be called interactively (see Section 22.3 [Interactive Call], page 433).

The return value of `defun` is undefined.

Here are some examples:

```
(defun foo () 5)
(foo)
     ⇒ 5
```

```
(defun bar (a &optional b &rest c)
    (list a b c))
(bar 1 2 3 4 5)
     ⇒ (1 2 (3 4 5))
(bar 1)
     ⇒ (1 nil nil)
(bar)
error  Wrong number of arguments.

(defun capitalize-backwards ()
  "Upcase the last letter of the word at point."
  (interactive)
  (backward-word 1)
  (forward-word 1)
  (backward-char 1)
  (capitalize-word 1))
```

Most Emacs functions are part of the source code of Lisp programs, and are defined when the Emacs Lisp reader reads the program source before executing it. However, you can also define functions dynamically at run time, e.g., by generating `defun` calls when your program's code is executed. If you do this, be aware that Emacs's Help commands, such as *C-h f*, which present in the `*Help*` buffer a button to jump to the function's definition, might be unable to find the source code because generating a function dynamically usually looks very different from the usual static calls to `defun`. You can make the job of finding the code which generates such functions easier by using the `definition-name` property, see Section 9.4.2 [Standard Properties], page 139.

Be careful not to redefine existing functions unintentionally. `defun` redefines even primitive functions such as `car` without any hesitation or notification. Emacs does not prevent you from doing this, because redefining a function is sometimes done deliberately, and there is no way to distinguish deliberate redefinition from unintentional redefinition.

`defalias` *name definition* **&optional** *doc* [Function]

This function defines the symbol *name* as a function, with definition *definition*. *definition* can be any valid Lisp function or macro, or a special form (see Section 10.1.7 [Special Forms], page 149), or a keymap (see Chapter 23 [Keymaps], page 481), or a vector or string (a keyboard macro). The return value of `defalias` is *undefined*.

If *doc* is non-`nil`, it becomes the function documentation of *name*. Otherwise, any documentation provided by *definition* is used.

Internally, `defalias` normally uses `fset` to set the definition. If *name* has a `defalias-fset-function` property, however, the associated value is used as a function to call in place of `fset`.

The proper place to use `defalias` is where a specific function or macro name is being defined—especially where that name appears explicitly in the source file being loaded. This is because `defalias` records which file defined the function, just like `defun` (see Section 16.9 [Unloading], page 314).

By contrast, in programs that manipulate function definitions for other purposes, it is better to use `fset`, which does not keep such records. See Section 13.9 [Function Cells], page 248.

`function-alias-p` *object* **&optional** *noerror* [Function]

Checks whether *object* is a function alias. If it is, it returns a list of symbols representing the function alias chain, else `nil`. For instance, if `a` is an alias for `b`, and `b` is an alias for `c`:

```
(function-alias-p 'a)
    ⇒ (b c)
```

If there's a loop in the definitions, an error will be signaled. If *noerror* is non-`nil`, the non-looping parts of the chain is returned instead.

You cannot create a new primitive function with `defun` or `defalias`, but you can use them to change the function definition of any symbol, even one such as `car` or `x-popup-menu` whose normal definition is a primitive. However, this is risky: for instance, it is next to impossible to redefine `car` without breaking Lisp completely. Redefining an obscure function such as `x-popup-menu` is less dangerous, but it still may not work as you expect. If there are calls to the primitive from C code, they call the primitive's C definition directly, so changing the symbol's definition will have no effect on them.

See also `defsubst`, which defines a function like `defun` and tells the Lisp compiler to perform inline expansion on it. See Section 13.14 [Inline Functions], page 261.

To undefine a function name, use `fmakunbound`. See Section 13.9 [Function Cells], page 248.

13.5 Calling Functions

Defining functions is only half the battle. Functions don't do anything until you *call* them, i.e., tell them to run. Calling a function is also known as *invocation*.

The most common way of invoking a function is by evaluating a list. For example, evaluating the list `(concat "a" "b")` calls the function `concat` with arguments `"a"` and `"b"`. See Chapter 10 [Evaluation], page 145, for a description of evaluation.

When you write a list as an expression in your program, you specify which function to call, and how many arguments to give it, in the text of the program. Usually that's just what you want. Occasionally you need to compute at run time which function to call. To do that, use the function `funcall`. When you also need to determine at run time how many arguments to pass, use `apply`.

`funcall` *function* **&rest** *arguments* [Function]

`funcall` calls *function* with *arguments*, and returns whatever *function* returns.

Since `funcall` is a function, all of its arguments, including *function*, are evaluated before `funcall` is called. This means that you can use any expression to obtain the function to be called. It also means that `funcall` does not see the expressions you write for the *arguments*, only their values. These values are *not* evaluated a second time in the act of calling *function*; the operation of `funcall` is like the normal procedure for calling a function, once its arguments have already been evaluated.

The argument *function* must be either a Lisp function or a primitive function. Special forms and macros are not allowed, because they make sense only when given the unevaluated argument expressions. `funcall` cannot provide these because, as we saw above, it never knows them in the first place.

If you need to use `funcall` to call a command and make it behave as if invoked interactively, use `funcall-interactively` (see Section 22.3 [Interactive Call], page 433).

```
(setq f 'list)
     ⇒ list
(funcall f 'x 'y 'z)
     ⇒ (x y z)
(funcall f 'x 'y '(z))
     ⇒ (x y (z))
(funcall 'and t nil)
error  Invalid function: #<subr and>
```

Compare these examples with the examples of `apply`.

`apply` *function* **&rest** *arguments* [Function]

`apply` calls *function* with *arguments*, just like `funcall` but with one difference: the last of *arguments* is a list of objects, which are passed to *function* as separate arguments, rather than a single list. We say that `apply` *spreads* this list so that each individual element becomes an argument.

`apply` with a single argument is special: the first element of the argument, which must be a non-empty list, is called as a function with the remaining elements as individual arguments. Passing two or more arguments will be faster.

`apply` returns the result of calling *function*. As with `funcall`, *function* must either be a Lisp function or a primitive function; special forms and macros do not make sense in `apply`.

```
(setq f 'list)
     ⇒ list
(apply f 'x 'y 'z)
error  Wrong type argument: listp, z
(apply '+ 1 2 '(3 4))
     ⇒ 10
(apply '+ '(1 2 3 4))
     ⇒ 10

(apply 'append '((a b c) nil (x y z) nil))
     ⇒ (a b c x y z)

(apply '(+ 3 4))
     ⇒ 7
```

For an interesting example of using `apply`, see [Definition of mapcar], page 242.

Sometimes it is useful to fix some of the function's arguments at certain values, and leave the rest of arguments for when the function is actually called. The act of fixing some

of the function's arguments is called *partial application* of the function[2]. The result is a new function that accepts the rest of arguments and calls the original function with all the arguments combined.

Here's how to do partial application in Emacs Lisp:

`apply-partially` *func* **&rest** *args* [Function]

This function returns a new function which, when called, will call *func* with the list of arguments composed from *args* and additional arguments specified at the time of the call. If *func* accepts *n* arguments, then a call to `apply-partially` with *m* `<=` *n* arguments will produce a new function of *n* `-` *m* arguments[3].

Here's how we could define the built-in function `1+`, if it didn't exist, using `apply-partially` and `+`, another built-in function[4]:

```
(defalias '1+ (apply-partially '+ 1)
  "Increment argument by one.")
(1+ 10)
     ⇒ 11
```

It is common for Lisp functions to accept functions as arguments or find them in data structures (especially in hook variables and property lists) and call them using `funcall` or `apply`. Functions that accept function arguments are often called *functionals*.

Sometimes, when you call a functional, it is useful to supply a no-op function as the argument. Here are three different kinds of no-op functions:

`identity` *argument* [Function]

This function returns *argument* and has no side effects.

`ignore` **&rest** *arguments* [Function]

This function ignores any *arguments* and returns `nil`.

`always` **&rest** *arguments* [Function]

This function ignores any *arguments* and returns `t`.

Some functions are user-visible *commands*, which can be called interactively (usually by a key sequence). It is possible to invoke such a command exactly as though it was called interactively, by using the `call-interactively` function. See Section 22.3 [Interactive Call], page 433.

13.6 Mapping Functions

A *mapping function* applies a given function (*not* a special form or macro) to each element of a list or other collection. Emacs Lisp has several such functions; this section describes `mapcar`, `mapc`, `mapconcat`, and `mapcan`, which map over a list. See [Definition of mapatoms],

[2] This is related to, but different from *currying*, which transforms a function that takes multiple arguments in such a way that it can be called as a chain of functions, each one with a single argument.

[3] If the number of arguments that *func* can accept is unlimited, then the new function will also accept an unlimited number of arguments, so in that case `apply-partially` doesn't reduce the number of arguments that the new function could accept.

[4] Note that unlike the built-in function, this version accepts any number of arguments.

page 137, for the function `mapatoms` which maps over the symbols in an obarray. See [Definition of maphash], page 130, for the function `maphash` which maps over key/value associations in a hash table.

These mapping functions do not allow char-tables because a char-table is a sparse array whose nominal range of indices is very large. To map over a char-table in a way that deals properly with its sparse nature, use the function `map-char-table` (see Section 6.6 [Char-Tables], page 119).

`mapcar` *function sequence* [Function]

`mapcar` applies *function* to each element of *sequence* in turn, and returns a list of the results.

The argument *sequence* can be any kind of sequence except a char-table; that is, a list, a vector, a bool-vector, or a string. The result is always a list. The length of the result is the same as the length of *sequence*. For example:

```
(mapcar #'car '((a b) (c d) (e f)))
     ⇒ (a c e)
(mapcar #'1+ [1 2 3])
     ⇒ (2 3 4)
(mapcar #'string "abc")
     ⇒ ("a" "b" "c")

;; Call each function in my-hooks.
(mapcar 'funcall my-hooks)

(defun mapcar* (function &rest args)
  "Apply FUNCTION to successive cars of all ARGS.
Return the list of results."
  ;; If no list is exhausted,
  (if (not (memq nil args))
      ;; apply function to CARs.
      (cons (apply function (mapcar #'car args))
            (apply #'mapcar* function
                   ;; Recurse for rest of elements.
                   (mapcar #'cdr args)))))

(mapcar* #'cons '(a b c) '(1 2 3 4))
     ⇒ ((a . 1) (b . 2) (c . 3))
```

`mapcan` *function sequence* [Function]

This function applies *function* to each element of *sequence*, like `mapcar`, but instead of collecting the results into a list, it returns a single list with all the elements of the results (which must be lists), by altering the results (using `nconc`; see Section 5.6.3 [Rearrangement], page 90). Like with `mapcar`, *sequence* can be of any type except a char-table.

```
;; Contrast this:
(mapcar #'list '(a b c d))
     ⇒ ((a) (b) (c) (d))
;; with this:
(mapcan #'list '(a b c d))
     ⇒ (a b c d)
```

`mapc` *function sequence* [Function]

`mapc` is like `mapcar` except that *function* is used for side-effects only—the values it returns are ignored, not collected into a list. `mapc` always returns *sequence*.

`mapconcat` *function sequence* **&optional** *separator* [Function]

`mapconcat` applies *function* to each element of *sequence*; the results, which must be sequences of characters (strings, vectors, or lists), are concatenated into a single string return value. Between each pair of result sequences, `mapconcat` inserts the characters from *separator*, which also must be a string, or a vector or list of characters; a `nil` value is treated as the empty string. See Chapter 6 [Sequences Arrays Vectors], page 102.

The argument *function* must be a function that can take one argument and returns a sequence of characters: a string, a vector, or a list. The argument *sequence* can be any kind of sequence except a char-table; that is, a list, a vector, a bool-vector, or a string.

```
(mapconcat #'symbol-name
           '(The cat in the hat)
           " ")
     ⇒ "The cat in the hat"

(mapconcat (lambda (x) (format "%c" (1+ x)))
           "HAL-8000")
     ⇒ "IBM.9111"
```

13.7 Anonymous Functions

Although functions are usually defined with `defun` and given names at the same time, it is sometimes convenient to use an explicit lambda expression—an *anonymous function*. Anonymous functions are valid wherever function names are. They are often assigned as variable values, or as arguments to functions; for instance, you might pass one as the *function* argument to `mapcar`, which applies that function to each element of a list (see Section 13.6 [Mapping Functions], page 241). See [describe-symbols example], page 599, for a realistic example of this.

When defining a lambda expression that is to be used as an anonymous function, you can in principle use any method to construct the list. But typically you should use the `lambda` macro, or the `function` special form, or the `#'` read syntax:

`lambda` *args* [*doc*] [*interactive*] *body*... [Macro]

This macro returns an anonymous function with argument list *args*, documentation string *doc* (if any), interactive spec *interactive* (if any), and body forms given by *body*.

Under dynamic binding, this macro effectively makes `lambda` forms self-quoting: evaluating a form whose CAR is `lambda` yields the form itself:

```
(lambda (x) (* x x))
     ⇒ (lambda (x) (* x x))
```

Note that when evaluating under lexical binding the result is a closure object (see Section 13.10 [Closures], page 250).

The `lambda` form has one other effect: it tells the Emacs evaluator and byte-compiler that its argument is a function, by using `function` as a subroutine (see below).

`function` *function-object* [Special Form]

This special form returns *function-object* without evaluating it. In this, it is similar to `quote` (see Section 10.2 [Quoting], page 151). But unlike `quote`, it also serves as a note to the Emacs evaluator and byte-compiler that *function-object* is intended to be used as a function. Assuming *function-object* is a valid lambda expression, this has two effects:

- When the code is byte-compiled, *function-object* is compiled into a byte-code function object (see Chapter 17 [Byte Compilation], page 317).
- When lexical binding is enabled, *function-object* is converted into a closure. See Section 13.10 [Closures], page 250.

When *function-object* is a symbol and the code is byte compiled, the byte-compiler will warn if that function is not defined or might not be known at run time.

The read syntax `#'` is a short-hand for using `function`. The following forms are all equivalent:

```
(lambda (x) (* x x))
(function (lambda (x) (* x x)))
#'(lambda (x) (* x x))
```

In the following example, we define a `change-property` function that takes a function as its third argument, followed by a `double-property` function that makes use of `change-property` by passing it an anonymous function:

```
(defun change-property (symbol prop function)
  (let ((value (get symbol prop)))
    (put symbol prop (funcall function value))))

(defun double-property (symbol prop)
  (change-property symbol prop (lambda (x) (* 2 x))))
```

Note that we do not quote the `lambda` form.

If you compile the above code, the anonymous function is also compiled. This would not happen if, say, you had constructed the anonymous function by quoting it as a list:

```
(defun double-property (symbol prop)
  (change-property symbol prop '(lambda (x) (* 2 x))))
```

In that case, the anonymous function is kept as a lambda expression in the compiled code. The byte-compiler cannot assume this list is a function, even though it looks like one, since it does not know that `change-property` intends to use it as a function.

13.8 Generic Functions

Functions defined using `defun` have a hard-coded set of assumptions about the types and expected values of their arguments. For example, a function that was designed to handle values of its argument that are either numbers or lists of numbers will fail or signal an error if called with a value of any other type, such as a vector or a string. This happens because the implementation of the function is not prepared to deal with types other than those assumed during the design.

By contrast, object-oriented programs use *polymorphic functions*: a set of specialized functions having the same name, each one of which was written for a certain specific set of argument types. Which of the functions is actually called is decided at run time based on the types of the actual arguments.

Emacs provides support for polymorphism. Like other Lisp environments, notably Common Lisp and its Common Lisp Object System (CLOS), this support is based on *generic functions*. The Emacs generic functions closely follow CLOS, including use of similar names, so if you have experience with CLOS, the rest of this section will sound very familiar.

A generic function specifies an abstract operation, by defining its name and list of arguments, but (usually) no implementation. The actual implementation for several specific classes of arguments is provided by *methods*, which should be defined separately. Each method that implements a generic function has the same name as the generic function, but the method's definition indicates what kinds of arguments it can handle by *specializing* the arguments defined by the generic function. These *argument specializers* can be more or less specific; for example, a `string` type is more specific than a more general type, such as `sequence`.

Note that, unlike in message-based OO languages, such as C++ and Simula, methods that implement generic functions don't belong to a class, they belong to the generic function they implement.

When a generic function is invoked, it selects the applicable methods by comparing the actual arguments passed by the caller with the argument specializers of each method. A method is applicable if the actual arguments of the call are compatible with the method's specializers. If more than one method is applicable, they are combined using certain rules, described below, and the combination then handles the call.

`cl-defgeneric` *name arguments* [*documentation*] [Macro]
[*options-and-methods...*] **&rest** *body*

This macro defines a generic function with the specified *name* and *arguments*. If *body* is present, it provides the default implementation. If *documentation* is present (it should always be), it specifies the documentation string for the generic function, in the form `(:documentation` *docstring*`)`. The optional *options-and-methods* can be one of the following forms:

`(declare` *declarations*`)`
: A declare form, as described in Section 13.15 [Declare Form], page 263.

`(:argument-precedence-order &rest` *args*`)`
: This form affects the sorting order for combining applicable methods. Normally, when two methods are compared during combination, method

arguments are examined left to right, and the first method whose argument specializer is more specific will come before the other one. The order defined by this form overrides that, and the arguments are examined according to their order in this form, and not left to right.

`(:method [`*qualifiers...*`] args &rest body)`
This form defines a method like `cl-defmethod` does.

`cl-defmethod` *name* [*extra*] [*qualifier*] *arguments* [**&context** (*expr* **spec**)...] **&rest** [*docstring*] *body* [Macro]

This macro defines a particular implementation for the generic function called *name*. The implementation code is given by *body*. If present, *docstring* is the documentation string for the method. The *arguments* list, which must be identical in all the methods that implement a generic function, and must match the argument list of that function, provides argument specializers of the form `(`*arg spec*`)`, where *arg* is the argument name as specified in the `cl-defgeneric` call, and *spec* is one of the following specializer forms:

type This specializer requires the argument to be of the given *type*, one of the types from the type hierarchy described below.

`(eql` *object*`)`
This specializer requires the argument be `eql` to the given *object*.

`(head` *object*`)`
The argument must be a cons cell whose `car` is `eql` to *object*.

struct-type
The argument must be an instance of a class named *struct-type* defined with `cl-defstruct` (see Section "Structures" in *Common Lisp Extensions for GNU Emacs Lisp*), or of one of its child classes.

Method definitions can make use of a new argument-list keyword, `&context`, which introduces extra specializers that test the environment at the time the method is run. This keyword should appear after the list of required arguments, but before any `&rest` or `&optional` keywords. The `&context` specializers look much like regular argument specializers—(*expr spec*)—except that *expr* is an expression to be evaluated in the current context, and the *spec* is a value to compare against. For example, `&context (overwrite-mode (eql t))` will make the method applicable only when `overwrite-mode` is turned on. The `&context` keyword can be followed by any number of context specializers. Because the context specializers are not part of the generic function's argument signature, they may be omitted in methods that don't require them.

The type specializer, `(`*arg type*`)`, can specify one of the *system types* in the following list. When a parent type is specified, an argument whose type is any of its more specific child types, as well as grand-children, grand-grand-children, etc. will also be compatible.

`integer` Parent type: `number`.

`number`

`null` Parent type: `symbol`

`symbol`

`string` Parent type: `array`.

`array` Parent type: `sequence`.

`cons` Parent type: `list`.

`list` Parent type: `sequence`.

`marker`

`overlay`

`float` Parent type: `number`.

`window-configuration`
`process`

`window`

`subr`

`compiled-function`
`buffer`

`char-table`
Parent type: `array`.

`bool-vector`
Parent type: `array`.

`vector` Parent type: `array`.

`frame`

`hash-table`
`font-spec`
`font-entity`
`font-object`

The optional *extra* element, expressed as '`:extra` *`string`*', allows you to add more methods, distinguished by *string*, for the same specializers and qualifiers.

The optional *qualifier* allows combining several applicable methods. If it is not present, the defined method is a *primary* method, responsible for providing the primary implementation of the generic function for the specialized arguments. You can also define *auxiliary methods*, by using one of the following values as *qualifier*:

`:before` This auxiliary method will run before the primary method. More accurately, all the `:before` methods will run before the primary, in the most-specific-first order.

`:after` This auxiliary method will run after the primary method. More accurately, all such methods will run after the primary, in the most-specific-last order.

`:around` This auxiliary method will run *instead* of the primary method. The most specific of such methods will be run before any other method. Such methods normally use `cl-call-next-method`, described below, to invoke the other auxiliary or primary methods.

Functions defined using `cl-defmethod` cannot be made interactive, i.e. commands (see Section 22.2 [Defining Commands], page 425), by adding the `interactive` form to them. If you need a polymorphic command, we recommend defining a normal command that calls a polymorphic function defined via `cl-defgeneric` and `cl-defmethod`.

Each time a generic function is called, it builds the *effective method* which will handle this invocation by combining the applicable methods defined for the function. The process of finding the applicable methods and producing the effective method is called *dispatch*. The applicable methods are those all of whose specializers are compatible with the actual arguments of the call. Since all of the arguments must be compatible with the specializers, they all determine whether a method is applicable. Methods that explicitly specialize more than one argument are called *multiple-dispatch methods*.

The applicable methods are sorted into the order in which they will be combined. The method whose left-most argument specializer is the most specific one will come first in the order. (Specifying `:argument-precedence-order` as part of `cl-defmethod` overrides that, as described above.) If the method body calls `cl-call-next-method`, the next most-specific method will run. If there are applicable `:around` methods, the most-specific of them will run first; it should call `cl-call-next-method` to run any of the less specific `:around` methods. Next, the `:before` methods run in the order of their specificity, followed by the primary method, and lastly the `:after` methods in the reverse order of their specificity.

`cl-call-next-method` **&rest** *args* [Function]
When invoked from within the lexical body of a primary or an `:around` auxiliary method, call the next applicable method for the same generic function. Normally, it is called with no arguments, which means to call the next applicable method with the same arguments that the calling method was invoked. Otherwise, the specified arguments are used instead.

`cl-next-method-p` [Function]
This function, when called from within the lexical body of a primary or an `:around` auxiliary method, returns non-`nil` if there is a next method to call.

13.9 Accessing Function Cell Contents

The *function definition* of a symbol is the object stored in the function cell of the symbol. The functions described here access, test, and set the function cell of symbols.

See also the function `indirect-function`. See [Definition of indirect-function], page 148.

`symbol-function` *symbol* [Function]
This returns the object in the function cell of *symbol*. It does not check that the returned object is a legitimate function.

If the function cell is void, the return value is `nil`. To distinguish between a function cell that is void and one set to `nil`, use `fboundp` (see below).

```
(defun bar (n) (+ n 2))
(symbol-function 'bar)
     ⇒ (lambda (n) (+ n 2))
(fset 'baz 'bar)
     ⇒ bar
(symbol-function 'baz)
     ⇒ bar
```

If you have never given a symbol any function definition, we say that that symbol's function cell is *void*. In other words, the function cell does not have any Lisp object in it. If you try to call the symbol as a function, Emacs signals a `void-function` error.

Note that void is not the same as `nil` or the symbol `void`. The symbols `nil` and `void` are Lisp objects, and can be stored into a function cell just as any other object can be (and `void` can be a valid function if you define it with `defun`). A void function cell contains no object whatsoever.

You can test the voidness of a symbol's function definition with `fboundp`. After you have given a symbol a function definition, you can make it void once more using `fmakunbound`.

`fboundp` *symbol* [Function]

This function returns `t` if the symbol has an object in its function cell, `nil` otherwise. It does not check that the object is a legitimate function.

`fmakunbound` *symbol* [Function]

This function makes *symbol*'s function cell void, so that a subsequent attempt to access this cell will cause a `void-function` error. It returns *symbol*. (See also `makunbound`, in Section 12.4 [Void Variables], page 193.)

```
(defun foo (x) x)
(foo 1)
     ⇒1
(fmakunbound 'foo)
     ⇒ foo
(foo 1)
error  Symbol's function definition is void: foo
```

`fset` *symbol definition* [Function]

This function stores *definition* in the function cell of *symbol*. The result is *definition*. Normally *definition* should be a function or the name of a function, but this is not checked. The argument *symbol* is an ordinary evaluated argument.

The primary use of this function is as a subroutine by constructs that define or alter functions, like `defun` or `advice-add` (see Section 13.12 [Advising Functions], page 252). You can also use it to give a symbol a function definition that is not a function, e.g., a keyboard macro (see Section 22.16 [Keyboard Macros], page 479):

```
;; Define a named keyboard macro.
(fset 'kill-two-lines "\^u2\^k")
     ⇒ "\^u2\^k"
```

If you wish to use `fset` to make an alternate name for a function, consider using `defalias` instead. See [Definition of defalias], page 238.

13.10 Closures

As explained in Section 12.10 [Variable Scoping], page 201, Emacs can optionally enable lexical binding of variables. When lexical binding is enabled, any named function that you create (e.g., with `defun`), as well as any anonymous function that you create using the `lambda` macro or the `function` special form or the `#'` syntax (see Section 13.7 [Anonymous Functions], page 243), is automatically converted into a *closure*.

A closure is a function that also carries a record of the lexical environment that existed when the function was defined. When it is invoked, any lexical variable references within its definition use the retained lexical environment. In all other respects, closures behave much like ordinary functions; in particular, they can be called in the same way as ordinary functions.

See Section 12.10.3 [Lexical Binding], page 203, for an example of using a closure.

Currently, an Emacs Lisp closure object is represented by a list with the symbol `closure` as the first element, a list representing the lexical environment as the second element, and the argument list and body forms as the remaining elements:

```
;; lexical binding is enabled.
(lambda (x) (* x x))
     ⇒ (closure (t) (x) (* x x))
```

However, the fact that the internal structure of a closure is exposed to the rest of the Lisp world is considered an internal implementation detail. For this reason, we recommend against directly examining or altering the structure of closure objects.

13.11 Open Closures

Traditionally, functions are opaque objects which offer no other functionality but to call them. (Emacs Lisp functions aren't fully opaque since you can extract some info out of them such as their docstring, their arglist, or their interactive spec, but they are still mostly opaque.) This is usually what we want, but occasionally we need functions to expose a bit more information about themselves.

Open closures, or *OClosures* for short, are function objects which carry additional type information and expose some information about themselves in the form of slots which you can access via accessor functions.

OClosures are defined in two steps: first you use `oclosure-define` to define a new OClosure type by specifying the slots carried by the OClosures of this type, and then you use `oclosure-lambda` to create an OClosure object of a given type.

Let's say we want to define keyboard macros, i.e. interactive functions which re-execute a sequence of key events (see Section 22.16 [Keyboard Macros], page 479). You could do it with a plain function as follows:

```
(defun kbd-macro (key-sequence)
  (lambda (&optional arg)
    (interactive "P")
    (execute-kbd-macro key-sequence arg)))
```

But with such a definition there is no easy way to extract the *key-sequence* from that function, for example to print it.

We can solve this problem using OClosures as follows. First we define the type of our keyboard macros (to which we decided to add a `counter` slot while at it):

```
(oclosure-define kbd-macro
  "Keyboard macro."
  keys (counter :mutable t))
```

After which we can rewrite our `kbd-macro` function:

```
(defun kbd-macro (key-sequence)
  (oclosure-lambda (kbd-macro (keys key-sequence) (counter 0))
      (&optional arg)
    (interactive "P")
    (execute-kbd-macro keys arg)
    (setq counter (1+ counter))))
```

As you can see, the `keys` and `counter` slots of the OClosure can be accessed as local variables from within the body of the OClosure. But we can now also access them from outside of the body of the OClosure, for example to describe a keyboard macro:

```
(defun describe-kbd-macro (km)
  (if (not (eq 'kbd-macro (oclosure-type km)))
      (message "Not a keyboard macro")
    (let ((keys    (kbd-macro--keys km))
          (counter (kbd-macro--counter km)))
      (message "Keys=%S, called %d times" keys counter))))
```

Where `kbd-macro--keys` and `kbd-macro--counter` are accessor functions generated by the `oclosure-define` macro for oclosures whose type is `kbd-macro`.

`oclosure-define` *oname* **&optional** *docstring* **&rest** *slots* [Macro]

This macro defines a new OClosure type along with accessor functions for its *slots*. *oname* can be a symbol (the name of the new type), or a list of the form (*oname* . *type-props*), in which case *type-props* is a list of additional properties of this oclosure type. *slots* is a list of slot descriptions where each slot can be either a symbol (the name of the slot) or it can be of the form (*slot-name* . *slot-props*), where *slot-props* is a property list of the corresponding slot *slot-name*. The OClosure type's properties specified by *type-props* can include the following:

`(:predicate` *pred-name*`)`

This requests creation of a predicate function named *pred-name*. This function will be used to recognize OClosures of the type *oname*. If this type property is not specified, `oclosure-define` will generate a default name for the predicate.

`(:parent` *otype*`)`

This makes type *otype* of OClosures be the parent of the type *oname*. The OClosures of type *oname* inherit the *slots* defined by their parent type.

`(:copier` *copier-name copier-args*`)`
: This causes the definition of a functional update function, knows as the *copier*, which takes an OClosure of type *oname* as its first argument and returns a copy of it with the slots named in *copier-args* modified to contain the value passed in the corresponding argument in the actual call to *copier-name*.

For each slot in *slots*, the `oclosure-define` macro creates an accessor function named *`oname--slot-name`*; these can be used to access the values of the slots. The slot definitions in *slots* can specify the following properties of the slots:

`:mutable` *val*
: By default, slots are immutable, but if you specify the `:mutable` property with a non-`nil` value, the slot can be mutated, for example with `setf` (see Section 12.17.1 [Setting Generalized Variables], page 224).

`:type` *val-type*
: This specifies the type of the values expected to appear in the slot.

`oclosure-lambda` (*type . slots*) *arglist* **&rest** *body* [Macro]
: This macro creates an anonymous OClosure of type *type*, which should have been defined with `oclosure-define`. *slots* should be a list of elements of the form *`(slot-name expr)`*. At run time, each *expr* is evaluated, in order, after which the OClosure is created with its slots initialized with the resulting values.

  When called as a function (see Section 13.5 [Calling Functions], page 239), the OClosure created by this macro will accept arguments according to *arglist* and will execute the code in *body*. *body* can refer to the value of any of its slot directly as if it were a local variable that had been captured by static scoping.

`oclosure-type` *object* [Function]
: This function returns the OClosure type (a symbol) of *object* if it is an OClosure, and `nil` otherwise.

One other function related to OClosures is `oclosure-interactive-form`, which allows some types of OClosures to compute their interactive forms dynamically. See Section 22.2.1 [Using Interactive], page 425.

13.12 Advising Emacs Lisp Functions

When you need to modify a function defined in another library, or when you need to modify a hook like *`foo`*`-function`, a process filter, or basically any variable or object field which holds a function value, you can use the appropriate setter function, such as `fset` or `defun` for named functions, `setq` for hook variables, or `set-process-filter` for process filters, but those are often too blunt, completely throwing away the previous value.

The *advice* feature lets you add to the existing definition of a function, by *advising the function*. This is a cleaner method than redefining the whole function.

Emacs's advice system provides two sets of primitives for that: the core set, for function values held in variables and object fields (with the corresponding primitives being `add-function` and `remove-function`) and another set layered on top of it for named functions (with the main primitives being `advice-add` and `advice-remove`).

As a trivial example, here's how to add advice that'll modify the return value of a function every time it's called:

```
(defun my-double (x)
  (* x 2))
(defun my-increase (x)
  (+ x 1))
(advice-add 'my-double :filter-return #'my-increase)
```

After adding this advice, if you call `my-double` with '3', the return value will be '7'. To remove this advice, say

```
(advice-remove 'my-double #'my-increase)
```

A more advanced example would be to trace the calls to the process filter of a process *proc*:

```
(defun my-tracing-function (proc string)
  (message "Proc %S received %S" proc string))

(add-function :before (process-filter proc) #'my-tracing-function)
```

This will cause the process's output to be passed to `my-tracing-function` before being passed to the original process filter. `my-tracing-function` receives the same arguments as the original function. When you're done with it, you can revert to the untraced behavior with:

```
(remove-function (process-filter proc) #'my-tracing-function)
```

Similarly, if you want to trace the execution of the function named `display-buffer`, you could use:

```
(defun his-tracing-function (orig-fun &rest args)
  (message "display-buffer called with args %S" args)
  (let ((res (apply orig-fun args)))
    (message "display-buffer returned %S" res)
    res))

(advice-add 'display-buffer :around #'his-tracing-function)
```

Here, `his-tracing-function` is called instead of the original function and receives the original function (additionally to that function's arguments) as argument, so it can call it if and when it needs to. When you're tired of seeing this output, you can revert to the untraced behavior with:

```
(advice-remove 'display-buffer #'his-tracing-function)
```

The arguments `:before` and `:around` used in the above examples specify how the two functions are composed, since there are many different ways to do it. The added function is also called a piece of *advice*.

13.12.1 Primitives to manipulate advices

`add-function` *where place function* **&optional** *props* [Macro]

This macro is the handy way to add the advice *function* to the function stored in *place* (see Section 12.17 [Generalized Variables], page 224).

where determines how *function* is composed with the existing function, e.g., whether *function* should be called before, or after the original function. See Section 13.12.3 [Advice Combinators], page 257, for the list of available ways to compose the two functions.

When modifying a variable (whose name will usually end with `-function`), you can choose whether *function* is used globally or only in the current buffer: if *place* is just a symbol, then *function* is added to the global value of *place*. Whereas if *place* is of the form `(local symbol)`, where *symbol* is an expression which returns the variable name, then *function* will only be added in the current buffer. Finally, if you want to modify a lexical variable, you will have to use `(var variable)`.

Every function added with `add-function` can be accompanied by an association list of properties *props*. Currently only two of those properties have a special meaning:

`name` This gives a name to the advice, which `remove-function` can use to identify which function to remove. Typically used when *function* is an anonymous function.

`depth` This specifies how to order the advice, should several pieces of advice be present. By default, the depth is 0. A depth of 100 indicates that this piece of advice should be kept as deep as possible, whereas a depth of −100 indicates that it should stay as the outermost piece. When two pieces of advice specify the same depth, the most recently added one will be outermost.

For `:before` advice, being outermost means that this advice will be run first, before any other advice, whereas being innermost means that it will run right before the original function, with no other advice run between itself and the original function. Similarly, for `:after` advice innermost means that it will run right after the original function, with no other advice run in between, whereas outermost means that it will be run right at the end after all other advice. An innermost `:override` piece of advice will only override the original function and other pieces of advice will apply to it, whereas an outermost `:override` piece of advice will override not only the original function but all other advice applied to it as well.

If *function* is not interactive, then the combined function will inherit the interactive spec, if any, of the original function. Else, the combined function will be interactive and will use the interactive spec of *function*. One exception: if the interactive spec of *function* is a function (i.e., a `lambda` expression or an `fbound` symbol rather than an expression or a string), then the interactive spec of the combined function will be a call to that function with the interactive spec of the original function as sole argument. To interpret the spec received as argument, use `advice-eval-interactive-spec`.

Note: The interactive spec of *function* will apply to the combined function and should hence obey the calling convention of the combined function rather than that of *function*. In many cases, it makes no difference since they are identical, but it does matter for `:around`, `:filter-args`, and `:filter-return`, where *function* receives different arguments than the original function stored in *place*.

`remove-function` *place function* [Macro]
: This macro removes *function* from the function stored in *place*. This only works if *function* was added to *place* using `add-function`.

  function is compared with functions added to *place* using `equal`, to try and make it work also with lambda expressions. It is additionally compared also with the `name` property of the functions added to *place*, which can be more reliable than comparing lambda expressions using `equal`.

`advice-function-member-p` *advice function-def* [Function]
: Return non-`nil` if *advice* is already in *function-def*. Like for `remove-function` above, instead of *advice* being the actual function, it can also be the `name` of the piece of advice.

`advice-function-mapc` *f function-def* [Function]
: Call the function *f* for every piece of advice that was added to *function-def*. *f* is called with two arguments: the advice function and its properties.

`advice-eval-interactive-spec` *spec* [Function]
: Evaluate the interactive *spec* just like an interactive call to a function with such a spec would, and then return the corresponding list of arguments that was built. E.g., `(advice-eval-interactive-spec "r\nP")` will return a list of three elements, containing the boundaries of the region and the current prefix argument.

  For instance, if you want to make the `C-x m` (`compose-mail`) command prompt for a '`From:`' header, you could say something like this:

```
(defun my-compose-mail-advice (orig &rest args)
  "Read From: address interactively."
  (interactive
   (lambda (spec)
     (let* ((user-mail-address
             (completing-read "From: "
                              '("one.address@example.net"
                                "alternative.address@example.net")))
            (from (message-make-from user-full-name
                                     user-mail-address))
            (spec (advice-eval-interactive-spec spec)))
       ;; Put the From header into the OTHER-HEADERS argument.
       (push (cons 'From from) (nth 2 spec))
       spec)))
  (apply orig args))

(advice-add 'compose-mail :around #'my-compose-mail-advice)
```

13.12.2 Advising Named Functions

A common use of advice is for named functions and macros. You could just use `add-function` as in:

```
(add-function :around (symbol-function 'fun) #'his-tracing-function)
```

But you should use `advice-add` and `advice-remove` for that instead. This separate set of functions to manipulate pieces of advice applied to named functions, offers the following extra features compared to `add-function`: they know how to deal with macros and autoloaded functions, they let `describe-function` preserve the original docstring as well as document the added advice, and they let you add and remove advice before a function is even defined.

`advice-add` can be useful for altering the behavior of existing calls to an existing function without having to redefine the whole function. However, it can be a source of bugs, since existing callers to the function may assume the old behavior, and work incorrectly when the behavior is changed by advice. Advice can also cause confusion in debugging, if the person doing the debugging does not notice or remember that the function has been modified by advice.

For these reasons, advice should be reserved for the cases where you cannot modify a function's behavior in any other way. If it is possible to do the same thing via a hook, that is preferable (see Section 24.1 [Hooks], page 525). If you simply want to change what a particular key does, it may be better to write a new command, and remap the old command's key bindings to the new one (see Section 23.14 [Remapping Commands], page 504).

If you are writing code for release, for others to use, try to avoid including advice in it. If the function you want to advise has no hook to do the job, please talk with the Emacs developers about adding a suitable hook. Especially, Emacs's own source files should not put advice on functions in Emacs. (There are currently a few exceptions to this convention, but we aim to correct them.) It is generally cleaner to create a new hook in `foo`, and make `bar` use the hook, than to have `bar` put advice in `foo`.

Special forms (see Section 10.1.7 [Special Forms], page 149) cannot be advised, however macros can be advised, in much the same way as functions. Of course, this will not affect code that has already been macro-expanded, so you need to make sure the advice is installed before the macro is expanded.

It is possible to advise a primitive (see Section 13.1 [What Is a Function], page 230), but one should typically *not* do so, for two reasons. Firstly, some primitives are used by the advice mechanism, and advising them could cause an infinite recursion. Secondly, many primitives are called directly from C, and such calls ignore advice; hence, one ends up in a confusing situation where some calls (occurring from Lisp code) obey the advice and other calls (from C code) do not.

`define-advice` *symbol* (*where lambda-list* **&optional** *name depth*) **&rest** *body* [Macro]

This macro defines a piece of advice and adds it to the function named *symbol*. The advice is an anonymous function if *name* is `nil` or a function named `symbol@name`. See `advice-add` for explanation of other arguments.

`advice-add` *symbol where function* **&optional** *props* [Function]

Add the advice *function* to the named function *symbol*. *where* and *props* have the same meaning as for `add-function` (see Section 13.12.1 [Core Advising Primitives], page 253).

`advice-remove` *symbol function* [Function]
Remove the advice *function* from the named function *symbol*. *function* can also be the `name` of a piece of advice.

`advice-member-p` *function symbol* [Function]
Return non-`nil` if the advice *function* is already in the named function *symbol*. *function* can also be the `name` of a piece of advice.

`advice-mapc` *function symbol* [Function]
Call *function* for every piece of advice that was added to the named function *symbol*. *function* is called with two arguments: the advice function and its properties.

13.12.3 Ways to compose advice

Here are the different possible values for the *where* argument of `add-function` and `advice-add`, specifying how the advice *function* and the original function should be composed.

`:before` Call *function* before the old function. Both functions receive the same arguments, and the return value of the composition is the return value of the old function. More specifically, the composition of the two functions behaves like:

```
(lambda (&rest r) (apply function r) (apply oldfun r))
```

`(add-function :before` *funvar function*`)` is comparable for single-function hooks to `(add-hook '`*hookvar function*`)` for normal hooks.

`:after` Call *function* after the old function. Both functions receive the same arguments, and the return value of the composition is the return value of the old function. More specifically, the composition of the two functions behaves like:

```
(lambda (&rest r) (prog1 (apply oldfun r) (apply function r)))
```

`(add-function :after` *funvar function*`)` is comparable for single-function hooks to `(add-hook '`*hookvar function* `'append)` for normal hooks.

`:override`
This completely replaces the old function with the new one. The old function can of course be recovered if you later call `remove-function`.

`:around` Call *function* instead of the old function, but provide the old function as an extra argument to *function*. This is the most flexible composition. For example, it lets you call the old function with different arguments, or many times, or within a let-binding, or you can sometimes delegate the work to the old function and sometimes override it completely. More specifically, the composition of the two functions behaves like:

```
(lambda (&rest r) (apply function oldfun r))
```

`:before-while`
Call *function* before the old function and don't call the old function if *function* returns `nil`. Both functions receive the same arguments, and the return value of the composition is the return value of the old function. More specifically, the composition of the two functions behaves like:

```
(lambda (&rest r) (and (apply function r) (apply oldfun r)))
```

`(add-function :before-while` *funvar function*`)` is comparable for single-function hooks to `(add-hook '`*hookvar function*`)` when *hookvar* is run via `run-hook-with-args-until-failure`.

`:before-until`
: Call *function* before the old function and only call the old function if *function* returns `nil`. More specifically, the composition of the two functions behaves like:

```
(lambda (&rest r) (or (apply function r) (apply oldfun r)))
```

`(add-function :before-until` *funvar function*`)` is comparable for single-function hooks to `(add-hook '`*hookvar function*`)` when *hookvar* is run via `run-hook-with-args-until-success`.

`:after-while`
: Call *function* after the old function and only if the old function returned non-`nil`. Both functions receive the same arguments, and the return value of the composition is the return value of *function*. More specifically, the composition of the two functions behaves like:

```
(lambda (&rest r) (and (apply oldfun r) (apply function r)))
```

`(add-function :after-while` *funvar function*`)` is comparable for single-function hooks to `(add-hook '`*hookvar function* `'append)` when *hookvar* is run via `run-hook-with-args-until-failure`.

`:after-until`
: Call *function* after the old function and only if the old function returned `nil`. More specifically, the composition of the two functions behaves like:

```
(lambda (&rest r) (or  (apply oldfun r) (apply function r)))
```

`(add-function :after-until` *funvar function*`)` is comparable for single-function hooks to `(add-hook '`*hookvar function* `'append)` when *hookvar* is run via `run-hook-with-args-until-success`.

`:filter-args`
: Call *function* first and use the result (which should be a list) as the new arguments to pass to the old function. More specifically, the composition of the two functions behaves like:

```
(lambda (&rest r) (apply oldfun (funcall function r)))
```

`:filter-return`
: Call the old function first and pass the result to *function*. More specifically, the composition of the two functions behaves like:

```
(lambda (&rest r) (funcall function (apply oldfun r)))
```

13.12.4 Adapting code using the old defadvice

A lot of code uses the old `defadvice` mechanism, which is largely made obsolete by the new `advice-add`, whose implementation and semantics is significantly simpler.

An old piece of advice such as:

```
(defadvice previous-line (before next-line-at-end
```

```
                                    (&optional arg try-vscroll))
  "Insert an empty line when moving up from the top line."
  (if (and next-line-add-newlines (= arg 1)
           (save-excursion (beginning-of-line) (bobp)))
      (progn
        (beginning-of-line)
        (newline))))
```

could be translated in the new advice mechanism into a plain function:

```
(defun previous-line--next-line-at-end (&optional arg try-vscroll)
  "Insert an empty line when moving up from the top line."
  (if (and next-line-add-newlines (= arg 1)
           (save-excursion (beginning-of-line) (bobp)))
      (progn
        (beginning-of-line)
        (newline))))
```

Obviously, this does not actually modify `previous-line`. For that the old advice needed:

```
(ad-activate 'previous-line)
```

whereas the new advice mechanism needs:

```
(advice-add 'previous-line :before #'previous-line--next-line-at-end)
```

Note that `ad-activate` had a global effect: it activated all pieces of advice enabled for that specified function. If you wanted to only activate or deactivate a particular piece, you needed to *enable* or *disable* it with `ad-enable-advice` and `ad-disable-advice`. The new mechanism does away with this distinction.

Around advice such as:

```
(defadvice foo (around foo-around)
  "Ignore case in `foo'."
  (let ((case-fold-search t))
    ad-do-it))
(ad-activate 'foo)
```

could translate into:

```
(defun foo--foo-around (orig-fun &rest args)
  "Ignore case in `foo'."
  (let ((case-fold-search t))
    (apply orig-fun args)))
(advice-add 'foo :around #'foo--foo-around)
```

Regarding the advice's *class*, note that the new `:before` is not quite equivalent to the old `before`, because in the old advice you could modify the function's arguments (e.g., with `ad-set-arg`), and that would affect the argument values seen by the original function, whereas in the new `:before`, modifying an argument via `setq` in the advice has no effect on the arguments seen by the original function. When porting `before` advice which relied on this behavior, you'll need to turn it into new `:around` or `:filter-args` advice instead.

Similarly old `after` advice could modify the returned value by changing `ad-return-value`, whereas new `:after` advice cannot, so when porting such old `after` advice, you'll need to turn it into new `:around` or `:filter-return` advice instead.

13.12.5 Advice and Byte Code

Not all functions can be reliably advised. The byte compiler may choose to replace a call to a function with a sequence of instructions that doesn't call the function you were interested in altering.

This usually happens due to one of the three following mechanisms:

`byte-compile` properties
: If a function's symbol has a `byte-compile` property, that property will be used instead of the symbol's function definition. See Section 17.2 [Compilation Functions], page 317.

`byte-optimize` properties
: If a function's symbol has a `byte-optimize` property, the byte compiler may rewrite the function arguments, or decide to use a different function altogether.

`compiler-macro` declare forms
: A function can have a special `compiler-macro declare` form in its definition (see Section 13.15 [Declare Form], page 263) that defines an *expander* to call when compiling the function. The expander could then cause the produced byte-code not to call the original function.

13.13 Declaring Functions Obsolete

You can mark a named function as *obsolete*, meaning that it may be removed at some point in the future. This causes Emacs to warn that the function is obsolete whenever it byte-compiles code containing that function, and whenever it displays the documentation for that function. In all other respects, an obsolete function behaves like any other function.

The easiest way to mark a function as obsolete is to put a `(declare (obsolete ...))` form in the function's `defun` definition. See Section 13.15 [Declare Form], page 263. Alternatively, you can use the `make-obsolete` function, described below.

A macro (see Chapter 14 [Macros], page 269) can also be marked obsolete with `make-obsolete`; this has the same effects as for a function. An alias for a function or macro can also be marked as obsolete; this makes the alias itself obsolete, not the function or macro which it resolves to.

`make-obsolete` *obsolete-name current-name when* [Function]
: This function marks *obsolete-name* as obsolete. *obsolete-name* should be a symbol naming a function or macro, or an alias for a function or macro.

  If *current-name* is a symbol, the warning message says to use *current-name* instead of *obsolete-name*. *current-name* does not need to be an alias for *obsolete-name*; it can be a different function with similar functionality. *current-name* can also be a string, which serves as the warning message. The message should begin in lower case, and end with a period. It can also be `nil`, in which case the warning message provides no additional details.

  The argument *when* should be a string indicating when the function was first made obsolete—for example, a date or a release number.

`define-obsolete-function-alias` *obsolete-name current-name when* **&optional** *doc* [Macro]

This convenience macro marks the function *obsolete-name* obsolete and also defines it as an alias for the function *current-name*. It is equivalent to the following:

```
(defalias obsolete-name current-name doc)
(make-obsolete obsolete-name current-name when)
```

In addition, you can mark a particular calling convention for a function as obsolete:

`set-advertised-calling-convention` *function signature when* [Function]

This function specifies the argument list *signature* as the correct way to call *function*. This causes the Emacs byte compiler to issue a warning whenever it comes across an Emacs Lisp program that calls *function* any other way (however, it will still allow the code to be byte compiled). *when* should be a string indicating when the variable was first made obsolete (usually a version number string).

For instance, in old versions of Emacs the `sit-for` function accepted three arguments, like this

```
(sit-for seconds milliseconds nodisp)
```

However, calling `sit-for` this way is considered obsolete (see Section 22.10 [Waiting], page 471). The old calling convention is deprecated like this:

```
(set-advertised-calling-convention
  'sit-for '(seconds &optional nodisp) "22.1")
```

The alternative to using this function is the `advertised-calling-convention` `declare` spec, see Section 13.15 [Declare Form], page 263.

13.14 Inline Functions

An *inline function* is a function that works just like an ordinary function, except for one thing: when you byte-compile a call to the function (see Chapter 17 [Byte Compilation], page 317), the function's definition is expanded into the caller.

The simple way to define an inline function, is to write `defsubst` instead of `defun`. The rest of the definition looks just the same, but using `defsubst` says to make it inline for byte compilation.

`defsubst` *name args* [*doc*] [*declare*] [*interactive*] *body...* [Macro]

This macro defines an inline function. Its syntax is exactly the same as `defun` (see Section 13.4 [Defining Functions], page 237).

Making a function inline often makes its function calls run faster. But it also has disadvantages. For one thing, it reduces flexibility; if you change the definition of the function, calls already inlined still use the old definition until you recompile them.

Another disadvantage is that making a large function inline can increase the size of compiled code both in files and in memory. Since the speed advantage of inline functions is greatest for small functions, you generally should not make large functions inline.

Also, inline functions do not behave well with respect to debugging, tracing, and advising (see Section 13.12 [Advising Functions], page 252). Since ease of debugging and the flexibility of redefining functions are important features of Emacs, you should not make a

function inline, even if it's small, unless its speed is really crucial, and you've timed the code to verify that using `defun` actually has performance problems.

After an inline function is defined, its inline expansion can be performed later on in the same file, just like macros.

It's possible to use `defmacro` to define a macro to expand into the same code that an inline function would execute (see Chapter 14 [Macros], page 269). But the macro would be limited to direct use in expressions—a macro cannot be called with `apply`, `mapcar` and so on. Also, it takes some work to convert an ordinary function into a macro. To convert it into an inline function is easy; just replace `defun` with `defsubst`. Since each argument of an inline function is evaluated exactly once, you needn't worry about how many times the body uses the arguments, as you do for macros.

Alternatively, you can define a function by providing the code which will inline it as a compiler macro (see Section 13.15 [Declare Form], page 263). The following macros make this possible.

`define-inline` *name args* [*doc*] [*declare*] *body...* [Macro]

Define a function *name* by providing code that does its inlining, as a compiler macro. The function will accept the argument list *args* and will have the specified *body*.

If present, *doc* should be the function's documentation string (see Section 13.2.4 [Function Documentation], page 235); *declare*, if present, should be a `declare` form (see Section 13.15 [Declare Form], page 263) specifying the function's metadata.

Functions defined via `define-inline` have several advantages with respect to macros defined by `defsubst` or `defmacro`:

- They can be passed to `mapcar` (see Section 13.6 [Mapping Functions], page 241).
- They are more efficient.
- They can be used as *place forms* to store values (see Section 12.17 [Generalized Variables], page 224).
- They behave in a more predictable way than `cl-defsubst` (see Section "Argument Lists" in *Common Lisp Extensions for GNU Emacs Lisp*).

Like `defmacro`, a function inlined with `define-inline` inherits the scoping rules, either dynamic or lexical, from the call site. See Section 12.10 [Variable Scoping], page 201.

The following macros should be used in the body of a function defined by `define-inline`.

`inline-quote` *expression* [Macro]

Quote *expression* for `define-inline`. This is similar to the backquote (see Section 10.3 [Backquote], page 151), but quotes code and accepts only `,`, not `,@`.

`inline-letevals` (*bindings...*) *body...* [Macro]

This provides a convenient way to ensure that the arguments to an inlined function are evaluated exactly once, as well as to create local variables.

It's similar to `let` (see Section 12.3 [Local Variables], page 190): It sets up local variables as specified by *bindings*, and then evaluates *body* with those bindings in effect.

Each element of *bindings* should be either a symbol or a list of the form `(var expr)`; the result is to evaluate *expr* and bind *var* to the result. However, when an element

of *bindings* is just a symbol *var*, the result of evaluating *var* is re-bound to *var* (which is quite different from the way `let` works).

The tail of *bindings* can be either `nil` or a symbol which should hold a list of arguments, in which case each argument is evaluated, and the symbol is bound to the resulting list.

`inline-const-p` *expression* [Macro]
Return non-`nil` if the value of *expression* is already known.

`inline-const-val` *expression* [Macro]
Return the value of *expression*.

`inline-error` *format* **&rest** *args* [Macro]
Signal an error, formatting *args* according to *format*.

Here's an example of using `define-inline`:

```
(define-inline myaccessor (obj)
  (inline-letevals (obj)
    (inline-quote (if (foo-p ,obj) (aref (cdr ,obj) 3) (aref ,obj 2)))))
```

This is equivalent to

```
(defsubst myaccessor (obj)
  (if (foo-p obj) (aref (cdr obj) 3) (aref obj 2)))
```

13.15 The `declare` Form

`declare` is a special macro which can be used to add meta properties to a function or macro: for example, marking it as obsolete, or giving its forms a special `TAB` indentation convention in Emacs Lisp mode.

`declare` *specs...* [Macro]
This macro ignores its arguments and evaluates to `nil`; it has no run-time effect. However, when a `declare` form occurs in the *declare* argument of a `defun` or `defsubst` function definition (see Section 13.4 [Defining Functions], page 237) or a `defmacro` macro definition (see Section 14.4 [Defining Macros], page 271), it appends the properties specified by *specs* to the function or macro. This work is specially performed by `defun`, `defsubst`, and `defmacro`.

Each element in *specs* should have the form `(`*`property args...`*`)`, which should not be quoted. These have the following effects:

`(advertised-calling-convention` *signature when*`)`
: This acts like a call to `set-advertised-calling-convention` (see Section 13.13 [Obsolete Functions], page 260); *signature* specifies the correct argument list for calling the function or macro, and *when* should be a string indicating when the old argument list was first made obsolete.

`(debug` *edebug-form-spec*`)`
: This is valid for macros only. When stepping through the macro with Edebug, use *edebug-form-spec*. See Section 19.2.15.1 [Instrumenting Macro Calls], page 359.

`(doc-string n)`
: This is used when defining a function or macro which itself will be used to define entities like functions, macros, or variables. It indicates that the *n*th argument, if any, should be considered as a documentation string.

`(indent indent-spec)`
: Indent calls to this function or macro according to *indent-spec*. This is typically used for macros, though it works for functions too. See Section 14.6 [Indenting Macros], page 276.

`(interactive-only value)`
: Set the function's `interactive-only` property to *value*. See [The interactive-only property], page 425.

`(obsolete current-name when)`
: Mark the function or macro as obsolete, similar to a call to `make-obsolete` (see Section 13.13 [Obsolete Functions], page 260). *current-name* should be a symbol (in which case the warning message says to use that instead), a string (specifying the warning message), or `nil` (in which case the warning message gives no extra details). *when* should be a string indicating when the function or macro was first made obsolete.

`(compiler-macro expander)`
: This can only be used for functions, and tells the compiler to use *expander* as an optimization function. When encountering a call to the function, of the form `(function args...)`, the macro expander will call *expander* with that form as well as with *args*..., and *expander* can either return a new expression to use instead of the function call, or it can return just the form unchanged, to indicate that the function call should be left alone.

  When *expander* is a lambda form it should be written with a single argument (i.e., be of the form `(lambda (arg) body)`) because the function's formal arguments are automatically added to the lambda's list of arguments for you.

`(gv-expander expander)`
: Declare *expander* to be the function to handle calls to the macro (or function) as a generalized variable, similarly to `gv-define-expander`. *expander* can be a symbol or it can be of the form `(lambda (arg) body)` in which case that function will additionally have access to the macro (or function)'s arguments.

`(gv-setter setter)`
: Declare *setter* to be the function to handle calls to the macro (or function) as a generalized variable. *setter* can be a symbol in which case it will be passed to `gv-define-simple-setter`, or it can be of the form `(lambda (arg) body)` in which case that function will additionally have access to the macro (or function)'s arguments and it will be passed to `gv-define-setter`.

`(completion` *completion-predicate*`)`
: Declare *completion-predicate* as a function to determine whether to include a function's symbol in the list of functions when asking for completions in `M-x`. This predicate function will only be called when `read-extended-command-predicate` is customized to `command-completion-default-include-p`; by default the value of `read-extended-command-predicate` is nil (see Section 22.3 [Interactive Call], page 433). The predicate *completion-predicate* is called with two arguments: the function's symbol and the current buffer.

`(modes` *modes*`)`
: Specify that this command is meant to be applicable only to specified *modes*. See Section 22.2.4 [Command Modes], page 431.

`(interactive-args` *arg* `...)`
: Specify the arguments that should be stored for `repeat-command`. Each *arg* is on the form *`argument-name form`*.

`(pure` *val*`)`
: If *val* is non-`nil`, this function is *pure* (see Section 13.1 [What Is a Function], page 230). This is the same as the `pure` property of the function's symbol (see Section 9.4.2 [Standard Properties], page 139).

`(side-effect-free` *val*`)`
: If *val* is non-`nil`, this function is free of side effects, so the byte compiler can ignore calls whose value is ignored. This is the same as the `side-effect-free` property of the function's symbol, see Section 9.4.2 [Standard Properties], page 139.

`(speed` *n*`)` Specify the value of `native-comp-speed` in effect for native compilation of this function (see Section 18.2 [Native-Compilation Variables], page 331). This allows function-level control of the optimization level used for native code emitted for the function. In particular, if *n* is -1, native compilation of the function will emit bytecode instead of native code for the function.

`no-font-lock-keyword`
: This is valid for macros only. Macros with this declaration are highlighted by font-lock (see Section 24.6 [Font Lock Mode], page 564) as normal functions, not specially as macros.

13.16 Telling the Compiler that a Function is Defined

Byte-compiling a file often produces warnings about functions that the compiler doesn't know about (see Section 17.6 [Compiler Errors], page 322). Sometimes this indicates a real problem, but usually the functions in question are defined in other files which would be loaded if that code is run. For example, byte-compiling `simple.el` used to warn:

```
simple.el:8727:1:Warning: the function ‘shell-mode’ is not known to be
    defined.
```

In fact, `shell-mode` is used only in a function that executes `(require 'shell)` before calling `shell-mode`, so `shell-mode` will be defined properly at run-time. When you know

that such a warning does not indicate a real problem, it is good to suppress the warning. That makes new warnings which might mean real problems more visible. You do that with `declare-function`.

All you need to do is add a `declare-function` statement before the first use of the function in question:

```
(declare-function shell-mode "shell" ())
```

This says that `shell-mode` is defined in `shell.el` (the '`.el`' can be omitted). The compiler takes for granted that that file really defines the function, and does not check.

The optional third argument specifies the argument list of `shell-mode`. In this case, it takes no arguments (`nil` is different from not specifying a value). In other cases, this might be something like `(file &optional overwrite)`. You don't have to specify the argument list, but if you do the byte compiler can check that the calls match the declaration.

`declare-function` *function file* **&optional** *arglist fileonly* [Macro]

Tell the byte compiler to assume that *function* is defined in the file *file*. The optional third argument *arglist* is either `t`, meaning the argument list is unspecified, or a list of formal parameters in the same style as `defun` (including the parentheses). An omitted *arglist* defaults to `t`, not `nil`; this is atypical behavior for omitted arguments, and it means that to supply a fourth but not third argument one must specify `t` for the third-argument placeholder instead of the usual `nil`. The optional fourth argument *fileonly* non-`nil` means check only that *file* exists, not that it actually defines *function*.

To verify that these functions really are declared where `declare-function` says they are, use `check-declare-file` to check all `declare-function` calls in one source file, or use `check-declare-directory` check all the files in and under a certain directory.

These commands find the file that ought to contain a function's definition using `locate-library`; if that finds no file, they expand the definition file name relative to the directory of the file that contains the `declare-function` call.

You can also say that a function is a primitive by specifying a file name ending in '`.c`' or '`.m`'. This is useful only when you call a primitive that is defined only on certain systems. Most primitives are always defined, so they will never give you a warning.

Sometimes a file will optionally use functions from an external package. If you prefix the filename in the `declare-function` statement with '`ext:`', then it will be checked if it is found, otherwise skipped without error.

There are some function definitions that '`check-declare`' does not understand (e.g., `defstruct` and some other macros). In such cases, you can pass a non-`nil` *fileonly* argument to `declare-function`, meaning to only check that the file exists, not that it actually defines the function. Note that to do this without having to specify an argument list, you should set the *arglist* argument to `t` (because `nil` means an empty argument list, as opposed to an unspecified one).

13.17 Determining whether a Function is Safe to Call

Some major modes, such as SES, call functions that are stored in user files. (See `ses`, for more information on SES.) User files sometimes have poor pedigrees—you can get a spreadsheet from someone you've just met, or you can get one through email from someone

you've never met. So it is risky to call a function whose source code is stored in a user file until you have determined that it is safe.

`unsafep` *form* **&optional** *unsafep-vars* [Function]

Returns `nil` if *form* is a *safe* Lisp expression, or returns a list that describes why it might be unsafe. The argument *unsafep-vars* is a list of symbols known to have temporary bindings at this point; it is mainly used for internal recursive calls. The current buffer is an implicit argument, which provides a list of buffer-local bindings.

Being quick and simple, `unsafep` does a very light analysis and rejects many Lisp expressions that are actually safe. There are no known cases where `unsafep` returns `nil` for an unsafe expression. However, a safe Lisp expression can return a string with a `display` property, containing an associated Lisp expression to be executed after the string is inserted into a buffer. This associated expression can be a virus. In order to be safe, you must delete properties from all strings calculated by user code before inserting them into buffers.

13.18 Other Topics Related to Functions

Here is a table of several functions that do things related to function calling and function definitions. They are documented elsewhere, but we provide cross references here.

`apply` See Section 13.5 [Calling Functions], page 239.

`autoload` See Section 16.5 [Autoload], page 305.

`call-interactively`
See Section 22.3 [Interactive Call], page 433.

`called-interactively-p`
See Section 22.4 [Distinguish Interactive], page 436.

`commandp` See Section 22.3 [Interactive Call], page 433.

`documentation`
See Section 25.2 [Accessing Documentation], page 598.

`eval` See Section 10.4 [Eval], page 152.

`funcall` See Section 13.5 [Calling Functions], page 239.

`function` See Section 13.7 [Anonymous Functions], page 243.

`ignore` See Section 13.5 [Calling Functions], page 239.

`indirect-function`
See Section 10.1.4 [Function Indirection], page 147.

`interactive`
See Section 22.2.1 [Using Interactive], page 425.

`interactive-p`
See Section 22.4 [Distinguish Interactive], page 436.

`mapatoms` See Section 9.3 [Creating Symbols], page 135.

`mapcar` See Section 13.6 [Mapping Functions], page 241.

`map-char-table`
: See Section 6.6 [Char-Tables], page 119.

`mapconcat`
: See Section 13.6 [Mapping Functions], page 241.

`undefined`
: See Section 23.11 [Functions for Key Lookup], page 496.

14 Macros

Macros enable you to define new control constructs and other language features. A macro is defined much like a function, but instead of telling how to compute a value, it tells how to compute another Lisp expression which will in turn compute the value. We call this expression the *expansion* of the macro.

Macros can do this because they operate on the unevaluated expressions for the arguments, not on the argument values as functions do. They can therefore construct an expansion containing these argument expressions or parts of them.

If you are using a macro to do something an ordinary function could do, just for the sake of speed, consider using an inline function instead. See Section 13.14 [Inline Functions], page 261.

14.1 A Simple Example of a Macro

Suppose we would like to define a Lisp construct to increment a variable value, much like the `++` operator in C. We would like to write `(inc x)` and have the effect of `(setq x (1+ x))`. Here's a macro definition that does the job:

```
(defmacro inc (var)
   (list 'setq var (list '1+ var)))
```

When this is called with `(inc x)`, the argument *var* is the symbol `x`—*not* the *value* of `x`, as it would be in a function. The body of the macro uses this to construct the expansion, which is `(setq x (1+ x))`. Once the macro definition returns this expansion, Lisp proceeds to evaluate it, thus incrementing `x`.

`macrop` *object* [Function]

This predicate tests whether its argument is a macro, and returns `t` if so, `nil` otherwise.

14.2 Expansion of a Macro Call

A macro call looks just like a function call in that it is a list which starts with the name of the macro. The rest of the elements of the list are the arguments of the macro.

Evaluation of the macro call begins like evaluation of a function call except for one crucial difference: the macro arguments are the actual expressions appearing in the macro call. They are not evaluated before they are given to the macro definition. By contrast, the arguments of a function are results of evaluating the elements of the function call list.

Having obtained the arguments, Lisp invokes the macro definition just as a function is invoked. The argument variables of the macro are bound to the argument values from the macro call, or to a list of them in the case of a `&rest` argument. And the macro body executes and returns its value just as a function body does.

The second crucial difference between macros and functions is that the value returned by the macro body is an alternate Lisp expression, also known as the *expansion* of the macro. The Lisp interpreter proceeds to evaluate the expansion as soon as it comes back from the macro.

Since the expansion is evaluated in the normal manner, it may contain calls to other macros. It may even be a call to the same macro, though this is unusual.

Note that Emacs tries to expand macros when loading an uncompiled Lisp file. This is not always possible, but if it is, it speeds up subsequent execution. See Section 16.1 [How Programs Do Loading], page 299.

You can see the expansion of a given macro call by calling `macroexpand`.

`macroexpand` *form* **&optional** *environment* [Function]

This function expands *form*, if it is a macro call. If the result is another macro call, it is expanded in turn, until something which is not a macro call results. That is the value returned by `macroexpand`. If *form* is not a macro call to begin with, it is returned as given.

Note that `macroexpand` does not look at the subexpressions of *form* (although some macro definitions may do so). Even if they are macro calls themselves, `macroexpand` does not expand them.

The function `macroexpand` does not expand calls to inline functions. Normally there is no need for that, since a call to an inline function is no harder to understand than a call to an ordinary function.

If *environment* is provided, it specifies an alist of macro definitions that shadow the currently defined macros. Byte compilation uses this feature.

```
(defmacro inc (var)
    (list 'setq var (list '1+ var)))

(macroexpand '(inc r))
     ⇒ (setq r (1+ r))

(defmacro inc2 (var1 var2)
    (list 'progn (list 'inc var1) (list 'inc var2)))

(macroexpand '(inc2 r s))
     ⇒ (progn (inc r) (inc s))  ; inc not expanded here.
```

`macroexpand-all` *form* **&optional** *environment* [Function]

`macroexpand-all` expands macros like `macroexpand`, but will look for and expand all macros in *form*, not just at the top-level. If no macros are expanded, the return value is `eq` to *form*.

Repeating the example used for `macroexpand` above with `macroexpand-all`, we see that `macroexpand-all` *does* expand the embedded calls to `inc`:

```
(macroexpand-all '(inc2 r s))
     ⇒ (progn (setq r (1+ r)) (setq s (1+ s)))
```

`macroexpand-1` *form* **&optional** *environment* [Function]

This function expands macros like `macroexpand`, but it only performs one step of the expansion: if the result is another macro call, `macroexpand-1` will not expand it.

14.3 Macros and Byte Compilation

You might ask why we take the trouble to compute an expansion for a macro and then evaluate the expansion. Why not have the macro body produce the desired results directly? The reason has to do with compilation.

When a macro call appears in a Lisp program being compiled, the Lisp compiler calls the macro definition just as the interpreter would, and receives an expansion. But instead of evaluating this expansion, it compiles the expansion as if it had appeared directly in the program. As a result, the compiled code produces the value and side effects intended for the macro, but executes at full compiled speed. This would not work if the macro body computed the value and side effects itself—they would be computed at compile time, which is not useful.

In order for compilation of macro calls to work, the macros must already be defined in Lisp when the calls to them are compiled. The compiler has a special feature to help you do this: if a file being compiled contains a `defmacro` form, the macro is defined temporarily for the rest of the compilation of that file.

Byte-compiling a file also executes any `require` calls at top-level in the file, so you can ensure that necessary macro definitions are available during compilation by requiring the files that define them (see Section 16.7 [Named Features], page 310). To avoid loading the macro definition files when someone *runs* the compiled program, write `eval-when-compile` around the `require` calls (see Section 17.5 [Eval During Compile], page 321).

14.4 Defining Macros

A Lisp macro object is a list whose CAR is `macro`, and whose CDR is a function. Expansion of the macro works by applying the function (with `apply`) to the list of *unevaluated* arguments from the macro call.

It is possible to use an anonymous Lisp macro just like an anonymous function, but this is never done, because it does not make sense to pass an anonymous macro to functionals such as `mapcar`. In practice, all Lisp macros have names, and they are almost always defined with the `defmacro` macro.

`defmacro` *name args* [*doc*] [*declare*] *body*... [Macro]

`defmacro` defines the symbol *name* (which should not be quoted) as a macro that looks like this:

```
(macro lambda args . body)
```

(Note that the CDR of this list is a lambda expression.) This macro object is stored in the function cell of *name*. The meaning of *args* is the same as in a function, and the keywords `&rest` and `&optional` may be used (see Section 13.2.3 [Argument List], page 234). Neither *name* nor *args* should be quoted. The return value of `defmacro` is undefined.

doc, if present, should be a string specifying the macro's documentation string. *declare*, if present, should be a `declare` form specifying metadata for the macro (see Section 13.15 [Declare Form], page 263). Note that macros cannot have interactive declarations, since they cannot be called interactively.

Macros often need to construct large list structures from a mixture of constants and nonconstant parts. To make this easier, use the ‘`’ syntax (see Section 10.3 [Backquote], page 151). For example:

```
(defmacro t-becomes-nil (variable)
  `(if (eq ,variable t)
       (setq ,variable nil)))

(t-becomes-nil foo)
     ≡ (if (eq foo t) (setq foo nil))
```

14.5 Common Problems Using Macros

Macro expansion can have counterintuitive consequences. This section describes some important consequences that can lead to trouble, and rules to follow to avoid trouble.

14.5.1 Wrong Time

The most common problem in writing macros is doing some of the real work prematurely—while expanding the macro, rather than in the expansion itself. For instance, one real package had this macro definition:

```
(defmacro my-set-buffer-multibyte (arg)
  (if (fboundp 'set-buffer-multibyte)
      (set-buffer-multibyte arg)))
```

With this erroneous macro definition, the program worked fine when interpreted but failed when compiled. This macro definition called `set-buffer-multibyte` during compilation, which was wrong, and then did nothing when the compiled package was run. The definition that the programmer really wanted was this:

```
(defmacro my-set-buffer-multibyte (arg)
  (if (fboundp 'set-buffer-multibyte)
      `(set-buffer-multibyte ,arg)))
```

This macro expands, if appropriate, into a call to `set-buffer-multibyte` that will be executed when the compiled program is actually run.

14.5.2 Evaluating Macro Arguments Repeatedly

When defining a macro you must pay attention to the number of times the arguments will be evaluated when the expansion is executed. The following macro (used to facilitate iteration) illustrates the problem. This macro allows us to write a for-loop construct.

```
(defmacro for (var from init to final do &rest body)
  "Execute a simple \"for\" loop.
For example, (for i from 1 to 10 do (print i))."
  (list 'let (list (list var init))
        (cons 'while
              (cons (list '<= var final)
                    (append body (list (list 'inc var)))))))
```

```
(for i from 1 to 3 do
   (setq square (* i i))
   (princ (format "\n%d %d" i square)))
↦
(let ((i 1))
  (while (<= i 3)
    (setq square (* i i))
    (princ (format "\n%d %d" i square))
    (inc i)))

     ⊣1       1
     ⊣2       4
     ⊣3       9
⇒ nil
```

The arguments `from`, `to`, and `do` in this macro are syntactic sugar; they are entirely ignored. The idea is that you will write noise words (such as `from`, `to`, and `do`) in those positions in the macro call.

Here's an equivalent definition simplified through use of backquote:

```
(defmacro for (var from init to final do &rest body)
  "Execute a simple \"for\" loop.
For example, (for i from 1 to 10 do (print i))."
  `(let ((,var ,init))
     (while (<= ,var ,final)
       ,@body
       (inc ,var))))
```

Both forms of this definition (with backquote and without) suffer from the defect that *final* is evaluated on every iteration. If *final* is a constant, this is not a problem. If it is a more complex form, say `(long-complex-calculation x)`, this can slow down the execution significantly. If *final* has side effects, executing it more than once is probably incorrect.

A well-designed macro definition takes steps to avoid this problem by producing an expansion that evaluates the argument expressions exactly once unless repeated evaluation is part of the intended purpose of the macro. Here is a correct expansion for the `for` macro:

```
(let ((i 1)
      (max 3))
  (while (<= i max)
    (setq square (* i i))
    (princ (format "%d      %d" i square))
    (inc i)))
```

Here is a macro definition that creates this expansion:

```
(defmacro for (var from init to final do &rest body)
  "Execute a simple for loop: (for i from 1 to 10 do (print i))."
  `(let ((,var ,init)
         (max ,final))
     (while (<= ,var max)
       ,@body
       (inc ,var))))
```

Unfortunately, this fix introduces another problem, described in the following section.

14.5.3 Local Variables in Macro Expansions

The new definition of `for` has a new problem: it introduces a local variable named `max` which the user does not expect. This causes trouble in examples such as the following:

```
(let ((max 0))
  (for x from 0 to 10 do
    (let ((this (frob x)))
      (if (< max this)
          (setq max this)))))
```

The references to `max` inside the body of the `for`, which are supposed to refer to the user's binding of `max`, really access the binding made by `for`.

The way to correct this is to use an uninterned symbol instead of `max` (see Section 9.3 [Creating Symbols], page 135). The uninterned symbol can be bound and referred to just like any other symbol, but since it is created by `for`, we know that it cannot already appear in the user's program. Since it is not interned, there is no way the user can put it into the program later. It will never appear anywhere except where put by `for`. Here is a definition of `for` that works this way:

```
(defmacro for (var from init to final do &rest body)
  "Execute a simple for loop: (for i from 1 to 10 do (print i))."
  (let ((tempvar (make-symbol "max")))
    `(let ((,var ,init)
           (,tempvar ,final))
       (while (<= ,var ,tempvar)
         ,@body
         (inc ,var)))))
```

This creates an uninterned symbol named `max` and puts it in the expansion instead of the usual interned symbol `max` that appears in expressions ordinarily.

14.5.4 Evaluating Macro Arguments in Expansion

Another problem can happen if the macro definition itself evaluates any of the macro argument expressions, such as by calling `eval` (see Section 10.4 [Eval], page 152). You have to take into account that macro expansion may take place long before the code is executed, when the context of the caller (where the macro expansion will be evaluated) is not yet accessible.

Also, if your macro definition does not use `lexical-binding`, its formal arguments may hide the user's variables of the same name. Inside the macro body, the macro argument

binding is the most local binding of such variable, so any references inside the form being evaluated do refer to it. Here is an example:

```
(defmacro foo (a)
  (list 'setq (eval a) t))
(setq x 'b)
(foo x) ↦ (setq b t)
     ⇒ t                    ; and b has been set.
;; but
(setq a 'c)
(foo a) ↦ (setq a t)
     ⇒ t                    ; but this set a, not c.
```

It makes a difference whether the user's variable is named `a` or `x`, because `a` conflicts with the macro argument variable `a`.

Also, the expansion of `(foo x)` above will return something different or signal an error when the code is compiled, since in that case `(foo x)` is expanded during compilation, whereas the execution of `(setq x 'b)` will only take place later when the code is executed.

To avoid these problems, **don't evaluate an argument expression while computing the macro expansion**. Instead, substitute the expression into the macro expansion, so that its value will be computed as part of executing the expansion. This is how the other examples in this chapter work.

14.5.5 How Many Times is the Macro Expanded?

Occasionally problems result from the fact that a macro call is expanded each time it is evaluated in an interpreted function, but is expanded only once (during compilation) for a compiled function. If the macro definition has side effects, they will work differently depending on how many times the macro is expanded.

Therefore, you should avoid side effects in computation of the macro expansion, unless you really know what you are doing.

One special kind of side effect can't be avoided: constructing Lisp objects. Almost all macro expansions include constructed lists; that is the whole point of most macros. This is usually safe; there is just one case where you must be careful: when the object you construct is part of a quoted constant in the macro expansion.

If the macro is expanded just once, in compilation, then the object is constructed just once, during compilation. But in interpreted execution, the macro is expanded each time the macro call runs, and this means a new object is constructed each time.

In most clean Lisp code, this difference won't matter. It can matter only if you perform side-effects on the objects constructed by the macro definition. Thus, to avoid trouble, **avoid side effects on objects constructed by macro definitions**. Here is an example of how such side effects can get you into trouble:

```
(defmacro empty-object ()
  (list 'quote (cons nil nil)))
```

```
(defun initialize (condition)
  (let ((object (empty-object)))
    (if condition
        (setcar object condition))
    object))
```

If `initialize` is interpreted, a new list `(nil)` is constructed each time `initialize` is called. Thus, no side effect survives between calls. If `initialize` is compiled, then the macro `empty-object` is expanded during compilation, producing a single constant `(nil)` that is reused and altered each time `initialize` is called.

One way to avoid pathological cases like this is to think of `empty-object` as a funny kind of constant, not as a memory allocation construct. You wouldn't use `setcar` on a constant such as `'(nil)`, so naturally you won't use it on `(empty-object)` either.

14.6 Indenting Macros

Within a macro definition, you can use the `declare` form (see Section 14.4 [Defining Macros], page 271) to specify how `TAB` should indent calls to the macro. An indentation specification is written like this:

```
(declare (indent indent-spec))
```

This results in the `lisp-indent-function` property being set on the macro name.

Here are the possibilities for *indent-spec*:

`nil` This is the same as no property—use the standard indentation pattern.

`defun` Handle this function like a '`def`' construct: treat the second line as the start of a *body*.

an integer, *number*
: The first *number* arguments of the function are *distinguished* arguments; the rest are considered the body of the expression. A line in the expression is indented according to whether the first argument on it is distinguished or not. If the argument is part of the body, the line is indented `lisp-body-indent` more columns than the open-parenthesis starting the containing expression. If the argument is distinguished and is either the first or second argument, it is indented *twice* that many extra columns. If the argument is distinguished and not the first or second argument, the line uses the standard pattern.

a symbol, *symbol*
: *symbol* should be a function name; that function is called to calculate the indentation of a line within this expression. The function receives two arguments:

  pos The position at which the line being indented begins.

  state The value returned by `parse-partial-sexp` (a Lisp primitive for indentation and nesting computation) when it parses up to the beginning of this line.

  It should return either a number, which is the number of columns of indentation for that line, or a list whose car is such a number. The difference between returning a number and returning a list is that a number says that all following

lines at the same nesting level should be indented just like this one; a list says that following lines might call for different indentations. This makes a difference when the indentation is being computed by `C-M-q`; if the value is a number, `C-M-q` need not recalculate indentation for the following lines until the end of the list.

15 Customization Settings

Users of Emacs can customize variables and faces without writing Lisp code, by using the Customize interface. See Section "Easy Customization" in *The GNU Emacs Manual*. This chapter describes how to define *customization items* that users can interact with through the Customize interface.

Customization items include customizable variables, which are defined with the `defcustom` macro; customizable faces, which are defined with `defface` (described separately in Section 41.12.2 [Defining Faces], page 1172); and *customization groups*, defined with `defgroup`, which act as containers for groups of related customization items.

15.1 Common Item Keywords

The customization declarations that we will describe in the next few sections—`defcustom`, `defgroup`, etc.—all accept keyword arguments (see Section 12.2 [Constant Variables], page 189) for specifying various information. This section describes keywords that apply to all types of customization declarations.

All of these keywords, except `:tag`, can be used more than once in a given item. Each use of the keyword has an independent effect. The keyword `:tag` is an exception because any given item can only display one name.

`:tag` *label*
: Use *label*, a string, instead of the item's name, to label the item in customization menus and buffers. **Don't use a tag which is substantially different from the item's real name; that would cause confusion.**

`:group` *group*
: Put this customization item in group *group*. If this keyword is missing from a customization item, it'll be placed in the same group that was last defined (in the current file).

  When you use `:group` in a `defgroup`, it makes the new group a subgroup of *group*.

  If you use this keyword more than once, you can put a single item into more than one group. Displaying any of those groups will show this item. Please don't overdo this, since the result would be annoying.

`:link` *link-data*
: Include an external link after the documentation string for this item. This is a sentence containing a button that references some other documentation.

  There are several alternatives you can use for *link-data*:

  `(custom-manual` *info-node*`)`
  : Link to an Info node; *info-node* is a string which specifies the node name, as in `"(emacs)Top"`. The link appears as '`[Manual]`' in the customization buffer and enters the built-in Info reader on *info-node*.

  `(info-link` *info-node*`)`
  : Like `custom-manual` except that the link appears in the customization buffer with the Info node name.

`(url-link` *`url`*`)`
: Link to a web page; *url* is a string which specifies the URL. The link appears in the customization buffer as *url* and invokes the WWW browser specified by `browse-url-browser-function`.

`(emacs-commentary-link` *`library`*`)`
: Link to the commentary section of a library; *library* is a string which specifies the library name. See Section D.8 [Library Headers], page 1348.

`(emacs-library-link` *`library`*`)`
: Link to an Emacs Lisp library file; *library* is a string which specifies the library name.

`(file-link` *`file`*`)`
: Link to a file; *file* is a string which specifies the name of the file to visit with `find-file` when the user invokes this link.

`(function-link` *`function`*`)`
: Link to the documentation of a function; *function* is a string which specifies the name of the function to describe with `describe-function` when the user invokes this link.

`(variable-link` *`variable`*`)`
: Link to the documentation of a variable; *variable* is a string which specifies the name of the variable to describe with `describe-variable` when the user invokes this link.

`(face-link` *`face`*`)`
: Link to the documentation of a face; *face* is a string which specifies the name of the face to describe with `describe-face` when the user invokes this link.

`(custom-group-link` *`group`*`)`
: Link to another customization group. Invoking it creates a new customization buffer for *group*.

You can specify the text to use in the customization buffer by adding `:tag` *`name`* after the first element of the *link-data*; for example, `(info-link :tag "foo" "(emacs)Top")` makes a link to the Emacs manual which appears in the buffer as '`foo`'.

You can use this keyword more than once, to add multiple links.

`:load` *`file`*
: Load file *file* (a string) before displaying this customization item (see Chapter 16 [Loading], page 299). Loading is done with `load`, and only if the file is not already loaded.

`:require` *`feature`*
: Execute `(require '`*`feature`*`)` when your saved customizations set the value of this item. *feature* should be a symbol.

The most common reason to use `:require` is when a variable enables a feature such as a minor mode, and just setting the variable won't have any effect unless the code which implements the mode is loaded.

`:version` *version*
: This keyword specifies that the item was first introduced in Emacs version *version*, or that its default value was changed in that version. The value *version* must be a string.

`:package-version '(`*package* `.` *version*`)`
: This keyword specifies that the item was first introduced in *package* version *version*, or that its meaning or default value was changed in that version. This keyword takes priority over `:version`.

  package should be the official name of the package, as a symbol (e.g., `MH-E`). *version* should be a string. If the package *package* is released as part of Emacs, *package* and *version* should appear in the value of `customize-package-emacs-version-alist`.

Packages distributed as part of Emacs that use the `:package-version` keyword must also update the `customize-package-emacs-version-alist` variable.

`customize-package-emacs-version-alist` [Variable]
: This alist provides a mapping for the versions of Emacs that are associated with versions of a package listed in the `:package-version` keyword. Its elements are:

  `(`*package* `(`*pversion* `.` *eversion*`)...)`

  For each *package*, which is a symbol, there are one or more elements that contain a package version *pversion* with an associated Emacs version *eversion*. These versions are strings. For example, the MH-E package updates this alist with the following:

```
(add-to-list 'customize-package-emacs-version-alist
             '(MH-E ("6.0" . "22.1") ("6.1" . "22.1") ("7.0" . "22.1")
                    ("7.1" . "22.1") ("7.2" . "22.1") ("7.3" . "22.1")
                    ("7.4" . "22.1") ("8.0" . "22.1")))
```

  The value of *package* needs to be unique and it needs to match the *package* value appearing in the `:package-version` keyword. Since the user might see the value in an error message, a good choice is the official name of the package, such as MH-E or Gnus.

15.2 Defining Customization Groups

Each Emacs Lisp package should have one main customization group which contains all the options, faces and other groups in the package. If the package has a small number of options and faces, use just one group and put everything in it. When there are more than twenty or so options and faces, then you should structure them into subgroups, and put the subgroups under the package's main customization group. It is OK to put some of the options and faces in the package's main group alongside the subgroups.

The package's main or only group should be a member of one or more of the standard customization groups. (To display the full list of them, use *M-x customize*.) Choose one or more of them (but not too many), and add your group to each of them using the `:group` keyword.

The way to declare new customization groups is with `defgroup`.

`defgroup` *group members doc* [*keyword value*]. . . [Macro]
Declare *group* as a customization group containing *members*. Do not quote the symbol *group*. The argument *doc* specifies the documentation string for the group.

The argument *members* is a list specifying an initial set of customization items to be members of the group. However, most often *members* is `nil`, and you specify the group's members by using the `:group` keyword when defining those members.

If you want to specify group members through *members*, each element should have the form `(`*`name widget`*`)`. Here *name* is a symbol, and *widget* is a widget type for editing that symbol. Useful widgets are `custom-variable` for a variable, `custom-face` for a face, and `custom-group` for a group.

When you introduce a new group into Emacs, use the `:version` keyword in the `defgroup`; then you need not use it for the individual members of the group.

In addition to the common keywords (see Section 15.1 [Common Keywords], page 278), you can also use this keyword in `defgroup`:

`:prefix` *prefix*
If the name of an item in the group starts with *prefix*, and the customizable variable `custom-unlispify-remove-prefixes` is non-`nil`, the item's tag will omit *prefix*. A group can have any number of prefixes.

The variables, faces, and subgroups of a group are stored in the `custom-group` property of the group's symbol. See Section 9.4.1 [Symbol Plists], page 138. The value of that property is a list of pairs whose `car` is the symbol of the variable or the face or the subgroup, and the `cdr` is one of the corresponding symbols `custom-variable`, `custom-face`, or `custom-group`.

`custom-unlispify-remove-prefixes` [User Option]
If this variable is non-`nil`, the prefixes specified by a group's `:prefix` keyword are omitted from tag names, whenever the user customizes the group.

The default value is `nil`, i.e., the prefix-discarding feature is disabled. This is because discarding prefixes often leads to confusing names for options and faces.

15.3 Defining Customization Variables

Customizable variables, also called *user options*, are global Lisp variables whose values can be set through the Customize interface. Unlike other global variables, which are defined with `defvar` (see Section 12.5 [Defining Variables], page 194), customizable variables are defined using the `defcustom` macro. In addition to calling `defvar` as a subroutine, `defcustom` states how the variable should be displayed in the Customize interface, the values it is allowed to take, etc.

`defcustom` *option standard doc* [*keyword value*]. . . [Macro]
This macro declares *option* as a user option (i.e., a customizable variable). You should not quote *option*.

The argument *standard* is an expression that specifies the standard value for *option*. Evaluating the `defcustom` form evaluates *standard*, but does not necessarily bind the

option to that value. If *option* already has a default value, it is left unchanged. If the user has already saved a customization for *option*, the user's customized value is installed as the default value. Otherwise, the result of evaluating *standard* is installed as the default value.

Like `defvar`, this macro marks `option` as a special variable, meaning that it should always be dynamically bound. If *option* is already lexically bound, that lexical binding remains in effect until the binding construct exits. See Section 12.10 [Variable Scoping], page 201.

The expression *standard* can be evaluated at various other times, too—whenever the customization facility needs to know *option*'s standard value. So be sure to use an expression which is harmless to evaluate at any time.

The argument *doc* specifies the documentation string for the variable.

If a `defcustom` does not specify any `:group`, the last group defined with `defgroup` in the same file will be used. This way, most `defcustom` do not need an explicit `:group`.

When you evaluate a `defcustom` form with `C-M-x` in Emacs Lisp mode (`eval-defun`), a special feature of `eval-defun` arranges to set the variable unconditionally, without testing whether its value is void. (The same feature applies to `defvar`, see Section 12.5 [Defining Variables], page 194.) Using `eval-defun` on a defcustom that is already defined calls the `:set` function (see below), if there is one.

If you put a `defcustom` in a pre-loaded Emacs Lisp file (see Section E.1 [Building Emacs], page 1351), the standard value installed at dump time might be incorrect, e.g., because another variable that it depends on has not been assigned the right value yet. In that case, use `custom-reevaluate-setting`, described below, to re-evaluate the standard value after Emacs starts up.

In addition to the keywords listed in Section 15.1 [Common Keywords], page 278, this macro accepts the following keywords:

`:type` *type*
: Use *type* as the data type for this option. It specifies which values are legitimate, and how to display the value (see Section 15.4 [Customization Types], page 285). Every `defcustom` should specify a value for this keyword.

`:options` *value-list*
: Specify the list of reasonable values for use in this option. The user is not restricted to using only these values, but they are offered as convenient alternatives.

  This is meaningful only for certain types, currently including `hook`, `plist` and `alist`. See the definition of the individual types for a description of how to use `:options`.

  Re-evaluating a `defcustom` form with a different `:options` value does not clear the values added by previous evaluations, or added by calls to `custom-add-frequent-value` (see below).

`:set` *setfunction*
: Specify *setfunction* as the way to change the value of this option when using the Customize interface. The function *setfunction* should take two arguments,

a symbol (the option name) and the new value, and should do whatever is necessary to update the value properly for this option (which may not mean simply setting the option as a Lisp variable); preferably, though, it should not modify its value argument destructively. The default for *setfunction* is `set-default-toplevel-value`.

If defined, *setfunction* will also be called when evaluating a `defcustom` form with `C-M-x` in Emacs Lisp mode and when the *option*'s value is changed via the `setopt` macro (see Section 12.8 [Setting Variables], page 198).

If you specify this keyword, the variable's documentation string should describe how to do the same job in hand-written Lisp code, either by invoking *setfunction* directly or by using `setopt`.

`:get` *getfunction*

Specify *getfunction* as the way to extract the value of this option. The function *getfunction* should take one argument, a symbol, and should return whatever customize should use as the current value for that symbol (which need not be the symbol's Lisp value). The default is `default-toplevel-value`.

You have to really understand the workings of Custom to use `:get` correctly. It is meant for values that are treated in Custom as variables but are not actually stored in Lisp variables. It is almost surely a mistake to specify *getfunction* for a value that really is stored in a Lisp variable.

`:initialize` *function*

function should be a function used to initialize the variable when the `defcustom` is evaluated. It should take two arguments, the option name (a symbol) and the value. Here are some predefined functions meant for use in this way:

`custom-initialize-set`

Use the variable's `:set` function to initialize the variable, but do not reinitialize it if it is already non-void.

`custom-initialize-default`

Like `custom-initialize-set`, but use the function `set-default-toplevel-value` to set the variable, instead of the variable's `:set` function. This is the usual choice for a variable whose `:set` function enables or disables a minor mode; with this choice, defining the variable will not call the minor mode function, but customizing the variable will do so.

`custom-initialize-reset`

Always use the `:set` function to initialize the variable. If the variable is already non-void, reset it by calling the `:set` function using the current value (returned by the `:get` method). This is the default `:initialize` function.

`custom-initialize-changed`

Use the `:set` function to initialize the variable, if it is already set or has been customized; otherwise, just use `set-default-toplevel-value`.

`custom-initialize-delay`
: This function behaves like `custom-initialize-set`, but it delays the actual initialization to the next Emacs start. This should be used in files that are preloaded (or for autoloaded variables), so that the initialization is done in the run-time context rather than the build-time context. This also has the side-effect that the (delayed) initialization is performed with the `:set` function. See Section E.1 [Building Emacs], page 1351.

`:local` *value*
: If the *value* is `t`, mark *option* as automatically buffer-local; if the value is `permanent`, also set *option*s `permanent-local` property to `t`. See Section 12.11.2 [Creating Buffer-Local], page 208.

`:risky` *value*
: Set the variable's `risky-local-variable` property to *value* (see Section 12.12 [File Local Variables], page 213).

`:safe` *function*
: Set the variable's `safe-local-variable` property to *function* (see Section 12.12 [File Local Variables], page 213).

`:set-after` *variables*
: When setting variables according to saved customizations, make sure to set the variables *variables* before this one; i.e., delay setting this variable until after those others have been handled. Use `:set-after` if setting this variable won't work properly unless those other variables already have their intended values.

It is useful to specify the `:require` keyword for an option that turns on a certain feature. This causes Emacs to load the feature, if it is not already loaded, whenever the option is set. See Section 15.1 [Common Keywords], page 278. Here is an example:

```
(defcustom frobnicate-automatically nil
  "Non-nil means automatically frobnicate all buffers."
  :type 'boolean
  :require 'frobnicate-mode
  :group 'frobnicate)
```

If a customization item has a type such as `hook` or `alist`, which supports `:options`, you can add additional values to the list from outside the `defcustom` declaration by calling `custom-add-frequent-value`. For example, if you define a function `my-lisp-mode-initialization` intended to be called from `emacs-lisp-mode-hook`, you might want to add that to the list of reasonable values for `emacs-lisp-mode-hook`, but not by editing its definition. You can do it thus:

```
(custom-add-frequent-value 'emacs-lisp-mode-hook
   'my-lisp-mode-initialization)
```

`custom-add-frequent-value` *symbol value* [Function]
: For the customization option *symbol*, add *value* to the list of reasonable values.

  The precise effect of adding a value depends on the customization type of *symbol*.

  Since evaluating a `defcustom` form does not clear values added previously, Lisp programs can use this function to add values for user options not yet defined.

Internally, `defcustom` uses the symbol property `standard-value` to record the expression for the standard value, `saved-value` to record the value saved by the user with the customization buffer, and `customized-value` to record the value set by the user with the customization buffer, but not saved. See Section 9.4 [Symbol Properties], page 138. In addition, there's `themed-value`, which is used to record the value set by a theme (see Section 15.6 [Custom Themes], page 296). These properties are lists, the car of which is an expression that evaluates to the value.

`custom-reevaluate-setting` *symbol* [Function]

This function re-evaluates the standard value of *symbol*, which should be a user option declared via `defcustom`. If the variable was customized, this function re-evaluates the saved value instead. Then it sets the user option to that value (using the option's `:set` property if that is defined).

This is useful for customizable options that are defined before their value could be computed correctly. For example, during startup Emacs calls this function for some user options that were defined in pre-loaded Emacs Lisp files, but whose initial values depend on information available only at run-time.

`custom-variable-p` *arg* [Function]

This function returns non-`nil` if *arg* is a customizable variable. A customizable variable is either a variable that has a `standard-value` or `custom-autoload` property (usually meaning it was declared with `defcustom`), or an alias for another customizable variable.

15.4 Customization Types

When you define a user option with `defcustom`, you must specify its *customization type*. That is a Lisp object which describes (1) which values are legitimate and (2) how to display the value in the customization buffer for editing.

You specify the customization type in `defcustom` with the `:type` keyword. The argument of `:type` is evaluated, but only once when the `defcustom` is executed, so it isn't useful for the value to vary. Normally we use a quoted constant. For example:

```
(defcustom diff-command "diff"
  "The command to use to run diff."
  :type '(string)
  :group 'diff)
```

In general, a customization type is a list whose first element is a symbol, one of the customization type names defined in the following sections. After this symbol come a number of arguments, depending on the symbol. Between the type symbol and its arguments, you can optionally write keyword-value pairs (see Section 15.4.4 [Type Keywords], page 292).

Some type symbols do not use any arguments; those are called *simple types*. For a simple type, if you do not use any keyword-value pairs, you can omit the parentheses around the type symbol. For example just `string` as a customization type is equivalent to `(string)`.

All customization types are implemented as widgets; see Section "Introduction" in *The Emacs Widget Library*, for details.

15.4.1 Simple Types

This section describes all the simple customization types. For several of these customization types, the customization widget provides inline completion with *C-M-i* or *M-*TAB.

`sexp` The value may be any Lisp object that can be printed and read back. You can use `sexp` as a fall-back for any option, if you don't want to take the time to work out a more specific type to use.

`integer` The value must be an integer.

`natnum` The value must be a nonnegative integer.

`number` The value must be a number (floating point or integer).

`float` The value must be floating point.

`string` The value must be a string. The customization buffer shows the string without delimiting '"' characters or '\' quotes.

`regexp` Like `string` except that the string must be a valid regular expression.

`character`
The value must be a character code. A character code is actually an integer, but this type shows the value by inserting the character in the buffer, rather than by showing the number.

`file` The value must be a file name. The widget provides completion.

`(file :must-match t)`
The value must be a file name for an existing file. The widget provides completion.

`directory`
The value must be a directory. The widget provides completion.

`hook` The value must be a list of functions. This customization type is used for hook variables. You can use the `:options` keyword in a hook variable's `defcustom` to specify a list of functions recommended for use in the hook; See Section 15.3 [Variable Definitions], page 281.

`symbol` The value must be a symbol. It appears in the customization buffer as the symbol name. The widget provides completion.

`function` The value must be either a lambda expression or a function name. The widget provides completion for function names.

`variable` The value must be a variable name. The widget provides completion.

`face` The value must be a symbol which is a face name. The widget provides completion.

`boolean` The value is boolean—either `nil` or `t`. Note that by using `choice` and `const` together (see the next section), you can specify that the value must be `nil` or `t`, but also specify the text to describe each value in a way that fits the specific meaning of the alternative.

`key` The value is a valid key according to *`key-valid-p`*, and suitable for use with, for example `keymap-set`.

`key-sequence`
The value is a key sequence. The customization buffer shows the key sequence using the same syntax as the *`kbd`* function. See Section 23.1 [Key Sequences], page 481. This is a legacy type; use `key` instead.

`coding-system`
The value must be a coding-system name, and you can do completion with *`M`*`-TAB`.

`color` The value must be a valid color name. The widget provides completion for color names, as well as a sample and a button for selecting a color name from a list of color names shown in a `*Colors*` buffer.

`fringe-bitmap`
The value must be a valid fringe bitmap name. The widget provides completion.

15.4.2 Composite Types

When none of the simple types is appropriate, you can use composite types, which build new types from other types or from specified data. The specified types or data are called the *arguments* of the composite type. The composite type normally looks like this:

```
(constructor arguments...)
```

but you can also add keyword-value pairs before the arguments, like this:

```
(constructor {keyword value}... arguments...)
```

Here is a table of constructors and how to use them to write composite types:

`(cons` *`car-type cdr-type`*`)`
The value must be a cons cell, its CAR must fit *car-type*, and its CDR must fit *cdr-type*. For example, `(cons string symbol)` is a customization type which matches values such as `("foo" . foo)`.

In the customization buffer, the CAR and CDR are displayed and edited separately, each according to their specified type.

`(list` *`element-types...`*`)`
The value must be a list with exactly as many elements as the *element-types* given; and each element must fit the corresponding *element-type*.

For example, `(list integer string function)` describes a list of three elements; the first element must be an integer, the second a string, and the third a function.

In the customization buffer, each element is displayed and edited separately, according to the type specified for it.

`(group` *`element-types...`*`)`
This works like `list` except for the formatting of text in the Custom buffer. `list` labels each element value with its tag; `group` does not.

`(vector` *`element-types...`*`)`
Like `list` except that the value must be a vector instead of a list. The elements work the same as in `list`.

`(alist :key-type` *key-type* `:value-type` *value-type*`)`

The value must be a list of cons-cells, the CAR of each cell representing a key of customization type *key-type*, and the CDR of the same cell representing a value of customization type *value-type*. The user can add and delete key/value pairs, and edit both the key and the value of each pair.

If omitted, *key-type* and *value-type* default to `sexp`.

The user can add any key matching the specified key type, but you can give some keys a preferential treatment by specifying them with the `:options` (see Section 15.3 [Variable Definitions], page 281). The specified keys will always be shown in the customize buffer (together with a suitable value), with a checkbox to include or exclude or disable the key/value pair from the alist. The user will not be able to edit the keys specified by the `:options` keyword argument.

The argument to the `:options` keywords should be a list of specifications for reasonable keys in the alist. Ordinarily, they are simply atoms, which stand for themselves. For example:

```
:options '("foo" "bar" "baz")
```

specifies that there are three known keys, namely `"foo"`, `"bar"` and `"baz"`, which will always be shown first.

You may want to restrict the value type for specific keys, for example, the value associated with the `"bar"` key can only be an integer. You can specify this by using a list instead of an atom in the list. The first element will specify the key, like before, while the second element will specify the value type. For example:

```
:options '("foo" ("bar" integer) "baz")
```

Finally, you may want to change how the key is presented. By default, the key is simply shown as a `const`, since the user cannot change the special keys specified with the `:options` keyword. However, you may want to use a more specialized type for presenting the key, like `function-item` if you know it is a symbol with a function binding. This is done by using a customization type specification instead of a symbol for the key.

```
:options '("foo"
           ((function-item some-function) integer)
           "baz")
```

Many alists use lists with two elements, instead of cons cells. For example,

```
(defcustom list-alist
  '(("foo" 1) ("bar" 2) ("baz" 3))
  "Each element is a list of the form (KEY VALUE).")
```

instead of

```
(defcustom cons-alist
  '(("foo" . 1) ("bar" . 2) ("baz" . 3))
  "Each element is a cons-cell (KEY . VALUE).")
```

Because of the way lists are implemented on top of cons cells, you can treat `list-alist` in the example above as a cons cell alist, where the value type is a list with a single element containing the real value.

```
(defcustom list-alist '(("foo" 1) ("bar" 2) ("baz" 3))
```

```
  "Each element is a list of the form (KEY VALUE)."
  :type '(alist :value-type (group integer)))
```

The `group` widget is used here instead of `list` only because the formatting is better suited for the purpose.

Similarly, you can have alists with more values associated with each key, using variations of this trick:

```
(defcustom person-data '(("brian"  50 t)
                         ("dorith" 55 nil)
                         ("ken"    52 t))
  "Alist of basic info about people.
Each element has the form (NAME AGE MALE-FLAG)."
  :type '(alist :value-type (group integer boolean)))
```

`(plist :key-type` *key-type* `:value-type` *value-type*`)`

This customization type is similar to `alist` (see above), except that (i) the information is stored as a property list, (see Section 5.9 [Property Lists], page 99), and (ii) *key-type*, if omitted, defaults to `symbol` rather than `sexp`.

`(choice` *alternative-types*`...)`

The value must fit one of *alternative-types*. For example, `(choice integer string)` allows either an integer or a string.

In the customization buffer, the user selects an alternative using a menu, and can then edit the value in the usual way for that alternative.

Normally the strings in this menu are determined automatically from the choices; however, you can specify different strings for the menu by including the `:tag` keyword in the alternatives. For example, if an integer stands for a number of spaces, while a string is text to use verbatim, you might write the customization type this way,

```
(choice (integer :tag "Number of spaces")
        (string :tag "Literal text"))
```

so that the menu offers '`Number of spaces`' and '`Literal text`'.

In any alternative for which `nil` is not a valid value, other than a `const`, you should specify a valid default for that alternative using the `:value` keyword. See Section 15.4.4 [Type Keywords], page 292.

If some values are covered by more than one of the alternatives, customize will choose the first alternative that the value fits. This means you should always list the most specific types first, and the most general last. Here's an example of proper usage:

```
(choice (const :tag "Off" nil)
        symbol (sexp :tag "Other"))
```

This way, the special value `nil` is not treated like other symbols, and symbols are not treated like other Lisp expressions.

`(radio` *element-types*`...)`

This is similar to `choice`, except that the choices are displayed using radio buttons rather than a menu. This has the advantage of displaying documenta-

tion for the choices when applicable and so is often a good choice for a choice between constant functions (`function-item` customization types).

`(const` *value*`)`
: The value must be *value*—nothing else is allowed.

  The main use of `const` is inside of `choice`. For example, `(choice integer (const nil))` allows either an integer or `nil`.

  `:tag` is often used with `const`, inside of `choice`. For example,

```
(choice (const :tag "Yes" t)
        (const :tag "No" nil)
        (const :tag "Ask" foo))
```

  describes a variable for which `t` means yes, `nil` means no, and `foo` means "ask".

`(other` *value*`)`
: This alternative can match any Lisp value, but if the user chooses this alternative, that selects the value *value*.

  The main use of `other` is as the last element of `choice`. For example,

```
(choice (const :tag "Yes" t)
        (const :tag "No" nil)
        (other :tag "Ask" foo))
```

  describes a variable for which `t` means yes, `nil` means no, and anything else means "ask". If the user chooses '`Ask`' from the menu of alternatives, that specifies the value `foo`; but any other value (not `t`, `nil` or `foo`) displays as '`Ask`', just like `foo`.

`(function-item` *function*`)`
: Like `const`, but used for values which are functions. This displays the documentation string as well as the function name. The documentation string is either the one you specify with `:doc`, or *function*'s own documentation string.

`(variable-item` *variable*`)`
: Like `const`, but used for values which are variable names. This displays the documentation string as well as the variable name. The documentation string is either the one you specify with `:doc`, or *variable*'s own documentation string.

`(set` *types...*`)`
: The value must be a list, and each element of the list must match one of the *types* specified.

  This appears in the customization buffer as a checklist, so that each of *types* may have either one corresponding element or none. It is not possible to specify two different elements that match the same one of *types*. For example, `(set integer symbol)` allows one integer and/or one symbol in the list; it does not allow multiple integers or multiple symbols. As a result, it is rare to use nonspecific types such as `integer` in a `set`.

  Most often, the *types* in a `set` are `const` types, as shown here:

```
(set (const :bold) (const :italic))
```

  Sometimes they describe possible elements in an alist:

```
(set (cons :tag "Height" (const height) integer)
```

```
(cons :tag "Width" (const width) integer))
```

That lets the user specify a height value optionally and a width value optionally.

`(repeat` *`element-type`*`)`
: The value must be a list and each element of the list must fit the type *element-type*. This appears in the customization buffer as a list of elements, with '`[INS]`' and '`[DEL]`' buttons for adding more elements or removing elements.

`(restricted-sexp :match-alternatives` *`criteria`*`)`
: This is the most general composite type construct. The value may be any Lisp object that satisfies one of *criteria*. *criteria* should be a list, and each element should be one of these possibilities:

  - A predicate—that is, a function of one argument that returns either `nil` or non-`nil` according to the argument. Using a predicate in the list says that objects for which the predicate returns non-`nil` are acceptable.
  - A quoted constant—that is, '*`object`*. This sort of element in the list says that *object* itself is an acceptable value.

  For example,

  ```
  (restricted-sexp :match-alternatives
                   (integerp 't 'nil))
  ```

  allows integers, `t` and `nil` as legitimate values.

  The customization buffer shows all legitimate values using their read syntax, and the user edits them textually.

Here is a table of the keywords you can use in keyword-value pairs in a composite type:

`:tag` *`tag`* Use *tag* as the name of this alternative, for user communication purposes. This is useful for a type that appears inside of a `choice`.

`:match-alternatives` *`criteria`*
: Use *criteria* to match possible values. This is used only in `restricted-sexp`.

`:args` *`argument-list`*
: Use the elements of *argument-list* as the arguments of the type construct. For instance, `(const :args (foo))` is equivalent to `(const foo)`. You rarely need to write `:args` explicitly, because normally the arguments are recognized automatically as whatever follows the last keyword-value pair.

15.4.3 Splicing into Lists

The `:inline` feature lets you splice a variable number of elements into the middle of a `list` or `vector` customization type. You use it by adding `:inline t` to a type specification which is contained in a `list` or `vector` specification.

Normally, each entry in a `list` or `vector` type specification describes a single element type. But when an entry contains `:inline t`, the value it matches is merged directly into the containing sequence. For example, if the entry matches a list with three elements, those become three elements of the overall sequence. This is analogous to '`,@`' in a backquote construct (see Section 10.3 [Backquote], page 151).

For example, to specify a list whose first element must be `baz` and whose remaining arguments should be zero or more of `foo` and `bar`, use this customization type:

```
(list (const baz) (set :inline t (const foo) (const bar)))
```

This matches values such as `(baz)`, `(baz foo)`, `(baz bar)` and `(baz foo bar)`.

When the element-type is a `choice`, you use `:inline` not in the `choice` itself, but in (some of) the alternatives of the `choice`. For example, to match a list which must start with a file name, followed either by the symbol `t` or two strings, use this customization type:

```
(list file
      (choice (const t)
              (list :inline t string string)))
```

If the user chooses the first alternative in the choice, then the overall list has two elements and the second element is `t`. If the user chooses the second alternative, then the overall list has three elements and the second and third must be strings.

The widgets can specify predicates to say whether an inline value matches the widget with the `:match-inline` element.

15.4.4 Type Keywords

You can specify keyword-argument pairs in a customization type after the type name symbol. Here are the keywords you can use, and their meanings:

`:value` *default*
: Provide a default value.

  If `nil` is not a valid value for the alternative, then it is essential to specify a valid default with `:value`.

  If you use this for a type that appears as an alternative inside of `choice`; it specifies the default value to use, at first, if and when the user selects this alternative with the menu in the customization buffer.

  Of course, if the actual value of the option fits this alternative, it will appear showing the actual value, not *default*.

`:format` *format-string*
: This string will be inserted in the buffer to represent the value corresponding to the type. The following '`%`' escapes are available for use in *format-string*:

  '`%[`*button*`%]`'
  : Display the text *button* marked as a button. The `:action` attribute specifies what the button will do if the user invokes it; its value is a function which takes two arguments—the widget which the button appears in, and the event.

    There is no way to specify two different buttons with different actions.

  '`%{`*sample*`%}`'
  : Show *sample* in a special face specified by `:sample-face`.

  '`%v`'
  : Substitute the item's value. How the value is represented depends on the kind of item, and (for variables) on the customization type.

‘`%d`’ Substitute the item's documentation string.

‘`%h`’ Like ‘`%d`’, but if the documentation string is more than one line, add a button to control whether to show all of it or just the first line.

‘`%t`’ Substitute the tag here. You specify the tag with the `:tag` keyword.

‘`%%`’ Display a literal ‘`%`’.

`:action` *action*
: Perform *action* if the user clicks on a button.

`:button-face` *face*
: Use the face *face* (a face name or a list of face names) for button text displayed with ‘`%[...%]`’.

`:button-prefix` *prefix*
`:button-suffix` *suffix*
: These specify the text to display before and after a button. Each can be:

  `nil` No text is inserted.

  a string The string is inserted literally.

  a symbol The symbol's value is used.

`:tag` *tag* Use *tag* (a string) as the tag for the value (or part of the value) that corresponds to this type.

`:doc` *doc* Use *doc* as the documentation string for this value (or part of the value) that corresponds to this type. In order for this to work, you must specify a value for `:format`, and use ‘`%d`’ or ‘`%h`’ in that value.

The usual reason to specify a documentation string for a type is to provide more information about the meanings of alternatives inside a `choice` type or the parts of some other composite type.

`:help-echo` *motion-doc*
: When you move to this item with `widget-forward` or `widget-backward`, it will display the string *motion-doc* in the echo area. In addition, *motion-doc* is used as the mouse `help-echo` string and may actually be a function or form evaluated to yield a help string. If it is a function, it is called with one argument, the widget.

`:match` *function*
: Specify how to decide whether a value matches the type. The corresponding value, *function*, should be a function that accepts two arguments, a widget and a value; it should return non-`nil` if the value is acceptable.

`:match-inline` *function*
: Specify how to decide whether an inline value matches the type. The corresponding value, *function*, should be a function that accepts two arguments, a widget and an inline value; it should return non-`nil` if the value is acceptable. See Section 15.4.3 [Splicing into Lists], page 291, for more information about inline values.

`:validate` *function*
: Specify a validation function for input. *function* takes a widget as an argument, and should return `nil` if the widget's current value is valid for the widget. Otherwise, it should return the widget containing the invalid data, and set that widget's `:error` property to a string explaining the error.

`:type-error` *string*
: *string* should be a string that describes why a value doesn't match the type, as determined by the `:match` function. When the `:match` function returns `nil`, the widget's `:error` property will be set to *string*.

15.4.5 Defining New Types

In the previous sections we have described how to construct elaborate type specifications for `defcustom`. In some cases you may want to give such a type specification a name. The obvious case is when you are using the same type for many user options: rather than repeat the specification for each option, you can give the type specification a name, and use that name each `defcustom`. The other case is when a user option's value is a recursive data structure. To make it possible for a datatype to refer to itself, it needs to have a name.

Since custom types are implemented as widgets, the way to define a new customize type is to define a new widget. We are not going to describe the widget interface here in details, see Section "Introduction" in *The Emacs Widget Library*, for that. Instead we are going to demonstrate the minimal functionality needed for defining new customize types by a simple example.

```
(define-widget 'binary-tree-of-string 'lazy
  "A binary tree made of cons-cells and strings."
  :offset 4
  :tag "Node"
  :type '(choice (string :tag "Leaf" :value "")
                 (cons :tag "Interior"
                       :value ("" . "")
                       binary-tree-of-string
                       binary-tree-of-string)))

(defcustom foo-bar ""
  "Sample variable holding a binary tree of strings."
  :type 'binary-tree-of-string)
```

The function to define a new widget is called `define-widget`. The first argument is the symbol we want to make a new widget type. The second argument is a symbol representing an existing widget, the new widget is going to be defined in terms of difference from the existing widget. For the purpose of defining new customization types, the `lazy` widget is perfect, because it accepts a `:type` keyword argument with the same syntax as the keyword argument to `defcustom` with the same name. The third argument is a documentation string for the new widget. You will be able to see that string with the *`M-x widget-browse`* `RET` *`binary-tree-of-string`* `RET` command.

After these mandatory arguments follow the keyword arguments. The most important is `:type`, which describes the data type we want to match with this widget. Here a

`binary-tree-of-string` is described as being either a string, or a cons-cell whose car and cdr are themselves both `binary-tree-of-string`. Note the reference to the widget type we are currently in the process of defining. The `:tag` attribute is a string to name the widget in the user interface, and the `:offset` argument is there to ensure that child nodes are indented four spaces relative to the parent node, making the tree structure apparent in the customization buffer.

The `defcustom` shows how the new widget can be used as an ordinary customization type.

The reason for the name `lazy` is that the other composite widgets convert their inferior widgets to internal form when the widget is instantiated in a buffer. This conversion is recursive, so the inferior widgets will convert *their* inferior widgets. If the data structure is itself recursive, this conversion is an infinite recursion. The `lazy` widget prevents the recursion: it convert its `:type` argument only when needed.

15.5 Applying Customizations

The following functions are responsible for installing the user's customization settings for variables and faces, respectively. When the user invokes '`Save for future sessions`' in the Customize interface, that takes effect by writing a `custom-set-variables` and/or a `custom-set-faces` form into the custom file, to be evaluated the next time Emacs starts.

`custom-set-variables` **&rest** *args* [Function]

This function installs the variable customizations specified by *args*. Each argument in *args* should have the form

```
(var expression [now [request [comment]]])
```

var is a variable name (a symbol), and *expression* is an expression which evaluates to the desired customized value.

If the `defcustom` form for *var* has been evaluated prior to this `custom-set-variables` call, *expression* is immediately evaluated, and the variable's value is set to the result. Otherwise, *expression* is stored into the variable's `saved-value` property, to be evaluated when the relevant `defcustom` is called (usually when the library defining that variable is loaded into Emacs).

The *now*, *request*, and *comment* entries are for internal use only, and may be omitted. *now*, if non-`nil`, means to set the variable's value now, even if the variable's `defcustom` form has not been evaluated. *request* is a list of features to be loaded immediately (see Section 16.7 [Named Features], page 310). *comment* is a string describing the customization.

`custom-set-faces` **&rest** *args* [Function]

This function installs the face customizations specified by *args*. Each argument in *args* should have the form

```
(face spec [now [comment]])
```

face is a face name (a symbol), and *spec* is the customized face specification for that face (see Section 41.12.2 [Defining Faces], page 1172).

The *now* and *comment* entries are for internal use only, and may be omitted. *now*, if non-`nil`, means to install the face specification now, even if the `defface` form has not been evaluated. *comment* is a string describing the customization.

15.6 Custom Themes

Custom themes are collections of settings that can be enabled or disabled as a unit. See Section "Custom Themes" in *The GNU Emacs Manual*. Each Custom theme is defined by an Emacs Lisp source file, which should follow the conventions described in this section. (Instead of writing a Custom theme by hand, you can also create one using a Customize-like interface; see Section "Creating Custom Themes" in *The GNU Emacs Manual*.)

A Custom theme file should be named `foo-theme.el`, where *foo* is the theme name. The first Lisp form in the file should be a call to `deftheme`, and the last form should be a call to `provide-theme`.

`deftheme` *theme* **&optional** *doc* **&rest** *properties* [Macro]

This macro declares *theme* (a symbol) as the name of a Custom theme. The optional argument *doc* should be a string describing the theme; this is the description shown when the user invokes the `describe-theme` command or types `?` in the '`*Custom Themes*`' buffer. The remaining arguments *properties* are used pass a property list with theme attributes.

The following attributes are supported:

`:family` A symbol designating what "family" a theme belongs to. A *family* of themes is a set of similar themes that differ by minor aspects, such as face colors that are meant for the light vs dark background of the frame.

`:kind` A symbol. If a theme is enabled and this property has the value `color-scheme`, then the `theme-choose-variant` command will look for other available themes that belong to the same family in order to switch the themes. Other values are currently unspecified and should not be used.

`:background-mode`
A symbol, either `light` or `dark`. This attribute is currently unused, but should still be specified.

Two special theme names are disallowed (using them causes an error): `user` is a dummy theme that stores the user's direct customization settings, and `changed` is a dummy theme that stores changes made outside of the Customize system.

`provide-theme` *theme* [Macro]

This macro declares that the theme named *theme* has been fully specified.

In between `deftheme` and `provide-theme` are Lisp forms specifying the theme settings: usually a call to `custom-theme-set-variables` and/or a call to `custom-theme-set-faces`.

`custom-theme-set-variables` *theme* **&rest** *args* [Function]

This function specifies the Custom theme *theme*'s variable settings. *theme* should be a symbol. Each argument in *args* should be a list of the form

```
(var expression [now [request [comment]]])
```

where the list entries have the same meanings as in `custom-set-variables`. See Section 15.5 [Applying Customizations], page 295.

`custom-theme-set-faces` *theme* **&rest** *args* [Function]
This function specifies the Custom theme *theme*'s face settings. *theme* should be a symbol. Each argument in *args* should be a list of the form

```
(face spec [now [comment]])
```

where the list entries have the same meanings as in `custom-set-faces`. See Section 15.5 [Applying Customizations], page 295.

In theory, a theme file can also contain other Lisp forms, which would be evaluated when loading the theme, but that is bad form. To protect against loading themes containing malicious code, Emacs displays the source file and asks for confirmation from the user before loading any non-built-in theme for the first time. As such, themes are not ordinarily byte-compiled, and source files usually take precedence when Emacs is looking for a theme to load.

The following functions are useful for programmatically enabling and disabling themes:

`custom-theme-p` *theme* [Function]
This function return a non-`nil` value if *theme* (a symbol) is the name of a Custom theme (i.e., a Custom theme which has been loaded into Emacs, whether or not the theme is enabled). Otherwise, it returns `nil`.

`custom-known-themes` [Variable]
The value of this variable is a list of themes loaded into Emacs. Each theme is represented by a Lisp symbol (the theme name). The default value of this variable is a list containing two dummy themes: `(user changed)`. The `changed` theme stores settings made before any Custom themes are applied (e.g., variables set outside of Customize). The `user` theme stores settings the user has customized and saved. Any additional themes declared with the `deftheme` macro are added to the front of this list.

`load-theme` *theme* **&optional** *no-confirm no-enable* [Command]
This function loads the Custom theme named *theme* from its source file, looking for the source file in the directories specified by the variable `custom-theme-load-path`. See Section "Custom Themes" in *The GNU Emacs Manual*. It also *enables* the theme (unless the optional argument *no-enable* is non-`nil`), causing its variable and face settings to take effect. It prompts the user for confirmation before loading the theme, unless the optional argument *no-confirm* is non-`nil`.

`require-theme` *feature* **&optional** *noerror* [Function]
This function searches `custom-theme-load-path` for a file that provides *feature* and then loads it. This is like the function `require` (see Section 16.7 [Named Features], page 310), except it searches `custom-theme-load-path` instead of `load-path` (see Section 16.3 [Library Search], page 302). This can be useful in Custom themes that need to load supporting Lisp files when `require` is unsuitable for that.

If *feature*, which should be a symbol, is not already present in the current Emacs session according to `featurep`, then `require-theme` searches for a file named *feature* with an added '`.elc`' or '`.el`' suffix, in that order, in the directories specified by `custom-theme-load-path`.

If a file providing *feature* is successfully found and loaded, then `require-theme` returns *feature*. The optional argument *noerror* determines what happens if the search or loading fails. If it is `nil`, the function signals an error; otherwise, it returns `nil`. If the file loads successfully but does not provide *feature*, then `require-theme` signals an error; this cannot be suppressed.

`enable-theme` *theme* [Command]

This function enables the Custom theme named *theme*. It signals an error if no such theme has been loaded.

`disable-theme` *theme* [Command]

This function disables the Custom theme named *theme*. The theme remains loaded, so that a subsequent call to `enable-theme` will re-enable it.

16 Loading

Loading a file of Lisp code means bringing its contents into the Lisp environment in the form of Lisp objects. Emacs finds and opens the file, reads the text, evaluates each form, and then closes the file. Such a file is also called a *Lisp library*.

The load functions evaluate all the expressions in a file just as the `eval-buffer` function evaluates all the expressions in a buffer. The difference is that the load functions read and evaluate the text in the file as found on disk, not the text in an Emacs buffer.

The loaded file must contain Lisp expressions, either as source code or as byte-compiled code. Each form in the file is called a *top-level form*. There is no special format for the forms in a loadable file; any form in a file may equally well be typed directly into a buffer and evaluated there. (Indeed, most code is tested this way.) Most often, the forms are function definitions and variable definitions.

Emacs can also load compiled dynamic modules: shared libraries that provide additional functionality for use in Emacs Lisp programs, just like a package written in Emacs Lisp would. When a dynamic module is loaded, Emacs calls a specially-named initialization function which the module needs to implement, and which exposes the additional functions and variables to Emacs Lisp programs.

For on-demand loading of external libraries which are known in advance to be required by certain Emacs primitives, see Section 42.21 [Dynamic Libraries], page 1304.

16.1 How Programs Do Loading

Emacs Lisp has several interfaces for loading. For example, `autoload` creates a placeholder object for a function defined in a file; trying to call the autoloading function loads the file to get the function's real definition (see Section 16.5 [Autoload], page 305). `require` loads a file if it isn't already loaded (see Section 16.7 [Named Features], page 310). Ultimately, all these facilities call the `load` function to do the work.

`load` *filename* **&optional** *missing-ok nomessage nosuffix must-suffix* [Function]
: This function finds and opens a file of Lisp code, evaluates all the forms in it, and closes the file.

  To find the file, `load` first looks for a file named *`filename`*`.elc`, that is, for a file whose name is *filename* with the extension '`.elc`' appended. If such a file exists, and Emacs was compiled with native-compilation support (see Chapter 18 [Native Compilation], page 328), `load` attempts to find a corresponding '`.eln`' file, and if found, loads it instead of *`filename`*`.elc`. Otherwise, it loads *`filename`*`.elc` (and starts a background native compilation to produce the missing '`.eln`' file, followed by loading that file). If there is no *`filename`*`.elc`, then `load` looks for a file named *`filename`*`.el`. If that file exists, it is loaded. If Emacs was compiled with support for dynamic modules (see Section 16.11 [Dynamic Modules], page 315), `load` next looks for a file named *`filename.ext`*, where *ext* is a system-dependent file-name extension of shared libraries ('`.so`' on GNU and Unix systems). Finally, if neither of those names is found, `load` looks for a file named *filename* with nothing appended, and loads it if it exists. (The `load` function is not clever about looking at *filename*. In the perverse case of a file named `foo.el.el`, evaluation of `(load "foo.el")` will indeed find it.)

If Auto Compression mode is enabled, as it is by default, then if `load` can not find a file, it searches for a compressed version of the file before trying other file names. It decompresses and loads it if it exists. It looks for compressed versions by appending each of the suffixes in `jka-compr-load-suffixes` to the file name. The value of this variable must be a list of strings. Its standard value is `(".gz")`.

If the optional argument *nosuffix* is non-`nil`, then `load` does not try the suffixes '`.elc`' and '`.el`'. In this case, you must specify the precise file name you want, except that, if Auto Compression mode is enabled, `load` will still use `jka-compr-load-suffixes` to find compressed versions. By specifying the precise file name and using `t` for *nosuffix*, you can prevent file names like `foo.el.el` from being tried.

If the optional argument *must-suffix* is non-`nil`, then `load` insists that the file name used must end in either '`.el`' or '`.elc`' (possibly extended with a compression suffix) or the shared-library extension, unless it contains an explicit directory name.

If the option `load-prefer-newer` is non-`nil`, then when searching suffixes, `load` selects whichever version of a file ('`.elc`', '`.el`', etc.) has been modified most recently. In this case, `load` doesn't load the '`.eln`' natively-compiled file even if it exists.

If *filename* is a relative file name, such as `foo` or `baz/foo.bar`, `load` searches for the file using the variable `load-path`. It appends *filename* to each of the directories listed in `load-path`, and loads the first file it finds whose name matches. The current default directory is tried only if it is specified in `load-path`, where `nil` stands for the default directory. `load` tries all three possible suffixes in the first directory in `load-path`, then all three suffixes in the second directory, and so on. See Section 16.3 [Library Search], page 302.

Whatever the name under which the file is eventually found, and the directory where Emacs found it, Emacs sets the value of the variable `load-file-name` to that file's name.

If you get a warning that `foo.elc` is older than `foo.el`, it means you should consider recompiling `foo.el`. See Chapter 17 [Byte Compilation], page 317.

When loading a source file (not compiled), `load` performs character set translation just as Emacs would do when visiting the file. See Section 34.10 [Coding Systems], page 980.

When loading an uncompiled file, Emacs tries to expand any macros that the file contains (see Chapter 14 [Macros], page 269). We refer to this as *eager macro expansion*. Doing this (rather than deferring the expansion until the relevant code runs) can significantly speed up the execution of uncompiled code. Sometimes, this macro expansion cannot be done, owing to a cyclic dependency. In the simplest example of this, the file you are loading refers to a macro defined in another file, and that file in turn requires the file you are loading. Emacs will issue an error about ('`Eager macro-expansion skipped due to cycle...`') giving details of the problem. You have to restructure your code so that this does not happen. Loading a compiled file does not cause macroexpansion, because this should already have happened during compilation. See Section 14.3 [Compiling Macros], page 271.

Messages like '`Loading foo...`' and '`Loading foo...done`' appear in the echo area during loading unless *nomessage* is non-`nil`. If a natively-compiled '`.eln`' file is loaded, the message says so.

Any unhandled errors while loading a file terminate loading. If the load was done for the sake of `autoload`, any function definitions made during the loading are undone.

If `load` can't find the file to load, then normally it signals a `file-error` (with '`Cannot open load file filename`'). But if *missing-ok* is non-`nil`, then `load` just returns `nil`.

You can use the variable `load-read-function` to specify a function for `load` to use instead of `read` for reading expressions. See below.

`load` returns `t` if the file loads successfully.

`load-file` *filename* [Command]
This command loads the file *filename*. If *filename* is a relative file name, then the current default directory is assumed. This command does not use `load-path`, and does not append suffixes. However, it does look for compressed versions (if Auto Compression Mode is enabled). Use this command if you wish to specify precisely the file name to load.

`load-library` *library* [Command]
This command loads the library named *library*. It is equivalent to `load`, except for the way it reads its argument interactively. See Section "Lisp Libraries" in *The GNU Emacs Manual*.

`load-in-progress` [Variable]
This variable is non-`nil` if Emacs is in the process of loading a file, and it is `nil` otherwise.

`load-file-name` [Variable]
When Emacs is in the process of loading a file, this variable's value is the name of that file, as Emacs found it during the search described earlier in this section.

`load-read-function` [Variable]
This variable specifies an alternate expression-reading function for `load` and `eval-region` to use instead of `read`. The function should accept one argument, just as `read` does.

By default, this variable's value is `read`. See Section 20.3 [Input Functions], page 375.

Instead of using this variable, it is cleaner to use another, newer feature: to pass the function as the *read-function* argument to `eval-region`. See [Eval], page 153.

For information about how `load` is used in building Emacs, see Section E.1 [Building Emacs], page 1351.

16.2 Load Suffixes

We now describe some technical details about the exact suffixes that `load` tries.

`load-suffixes` [Variable]
This is a list of suffixes indicating (compiled or source) Emacs Lisp files. It should not include the empty string. `load` uses these suffixes in order when it appends Lisp suffixes to the specified file name. The standard value is `(".elc" ".el")` which produces the behavior described in the previous section.

`load-file-rep-suffixes` [Variable]

This is a list of suffixes that indicate representations of the same file. This list should normally start with the empty string. When `load` searches for a file it appends the suffixes in this list, in order, to the file name, before searching for another file.

Enabling Auto Compression mode appends the suffixes in `jka-compr-load-suffixes` to this list and disabling Auto Compression mode removes them again. The standard value of `load-file-rep-suffixes` if Auto Compression mode is disabled is `("")`. Given that the standard value of `jka-compr-load-suffixes` is `(".gz")`, the standard value of `load-file-rep-suffixes` if Auto Compression mode is enabled is `("" ".gz")`.

`get-load-suffixes` [Function]

This function returns the list of all suffixes that `load` should try, in order, when its *must-suffix* argument is non-`nil`. This takes both `load-suffixes` and `load-file-rep-suffixes` into account. If `load-suffixes`, `jka-compr-load-suffixes` and `load-file-rep-suffixes` all have their standard values, this function returns `(".elc" ".elc.gz" ".el" ".el.gz")` if Auto Compression mode is enabled and `(".elc" ".el")` if Auto Compression mode is disabled.

To summarize, `load` normally first tries the suffixes in the value of `(get-load-suffixes)` and then those in `load-file-rep-suffixes`. If *nosuffix* is non-`nil`, it skips the former group, and if *must-suffix* is non-`nil`, it skips the latter group.

`load-prefer-newer` [User Option]

If this option is non-`nil`, then rather than stopping at the first suffix that exists, `load` tests them all, and uses whichever file is the newest.

16.3 Library Search

When Emacs loads a Lisp library, it searches for the library in a list of directories specified by the variable `load-path`.

`load-path` [Variable]

The value of this variable is a list of directories to search when loading files with `load`. Each element is a string (which must be a directory) or `nil` (which stands for the current working directory).

When Emacs starts up, it sets up the value of `load-path` in several steps. First, it looks for the directory containing its own Lisp files, using default locations set when Emacs was compiled. It saves this directory in `lisp-directory`. Normally, this is a directory where the `*.elc` files are installed, something like

`"/usr/local/share/emacs/`*version*`/lisp"`

where *version* is the Emacs version. (In this and the following examples, replace `/usr/local` with the prefix appropriate for your Emacs installation.) This directory and its subdirectories contain the standard Lisp files that come with Emacs. If Emacs cannot find its own Lisp files, it will not start correctly.

If you run Emacs from the directory where it was built—that is, an executable that has not been installed yet—Emacs instead initializes `lisp-directory` using the `lisp` subdirectory of the directory containing the sources from which it was built.

Emacs then initializes `load-path` with this `lisp-directory`. If you built Emacs in a separate directory from the sources, it also adds the `lisp` subdirectory of the build directory.

All of these directories are stored in the above two variables as absolute file names.

Unless you start Emacs with the `--no-site-lisp` option, it then adds two more `site-lisp` directories to the front of `load-path`. These are intended for locally installed Lisp files, and are normally of the form:

```
"/usr/local/share/emacs/version/site-lisp"
```

and

```
"/usr/local/share/emacs/site-lisp"
```

The first one is for locally installed files for the current Emacs *version*; the second is for locally installed files meant for use with any installed Emacs version. (If Emacs is running uninstalled, it also adds `site-lisp` subdirectories from the source and build directories, if they exist. However, normally the source and build directories do not contain `site-lisp` subdirectories.)

If the environment variable `EMACSLOADPATH` is set, it modifies the above initialization procedure. Emacs initializes `load-path` based on the value of the environment variable.

The syntax of `EMACSLOADPATH` is the same as used for `PATH`; directories are separated by ‘`:`’ (or ‘`;`’, on some operating systems). Here is an example of how to set `EMACSLOADPATH` variable (from a `sh`-style shell):

```
export EMACSLOADPATH=/home/foo/.emacs.d/lisp:
```

An empty element in the value of the environment variable, whether trailing (as in the above example, note the trailing ‘`:`’), leading, or embedded, is replaced by the default value of `load-path` as determined by the standard initialization procedure. If there are no such empty elements, then `EMACSLOADPATH` specifies the entire `load-path`. You must include either an empty element, or the explicit path to the directory containing the standard Lisp files, else Emacs will not function. (Another way to modify `load-path` is to use the `-L` command-line option when starting Emacs; see below.)

For each directory in `load-path`, Emacs then checks to see if it contains a file `subdirs.el`, and if so, loads it. The `subdirs.el` file is created when Emacs is built/installed, and contains code that causes Emacs to add any subdirectories of those directories to `load-path`. Both immediate subdirectories and subdirectories multiple levels down are added. But it excludes subdirectories whose names do not start with a letter or digit, and subdirectories named `RCS` or `CVS`, and subdirectories containing a file named `.nosearch`.

Next, Emacs adds any extra load directories that you specify using the `-L` command-line option (see Section “Action Arguments” in *The GNU Emacs Manual*). It also adds the directories where optional packages are installed, if any (see Section 43.1 [Packaging Basics], page 1308).

It is common to add code to one’s init file (see Section 42.1.2 [Init File], page 1263) to add one or more directories to `load-path`. For example:

```
(push "~/.emacs.d/lisp" load-path)
```

See Section 5.5 [List Variables], page 85, for the description of `push`.

Dumping Emacs uses a special value of `load-path`. If you use a `site-load.el` or `site-init.el` file to customize the dumped Emacs (see Section E.1 [Building Emacs], page 1351), any changes to `load-path` that these files make will be lost after dumping.

`lisp-directory` [Variable]
: This variable holds a string naming the directory which holds Emacs's own `*.el` and `*.elc` files. This is usually the place where those files are located in the Emacs installation tree, unless Emacs is run from its build directory in which case it points to the `lisp` subdirectory in the source directory from which Emacs was built.

`locate-library` *library* **&optional** *nosuffix path interactive-call* [Command]
: This command finds the precise file name for library *library*. It searches for the library in the same way `load` does, and the argument *nosuffix* has the same meaning as in `load`: don't add suffixes '`.elc`' or '`.el`' to the specified name *library*.

  If the *path* is non-`nil`, that list of directories is used instead of `load-path`.

  When `locate-library` is called from a program, it returns the file name as a string. When the user runs `locate-library` interactively, the argument *interactive-call* is `t`, and this tells `locate-library` to display the file name in the echo area.

`list-load-path-shadows` **&optional** *stringp* [Command]
: This command shows a list of *shadowed* Emacs Lisp files. A shadowed file is one that will not normally be loaded, despite being in a directory on `load-path`, due to the existence of another similarly-named file in a directory earlier on `load-path`.

  For instance, suppose `load-path` is set to

  ```
  ("/opt/emacs/site-lisp" "/usr/share/emacs/29.1/lisp")
  ```

  and that both these directories contain a file named `foo.el`. Then `(require 'foo)` never loads the file in the second directory. Such a situation might indicate a problem in the way Emacs was installed.

  When called from Lisp, this function prints a message listing the shadowed files, instead of displaying them in a buffer. If the optional argument `stringp` is non-`nil`, it instead returns the shadowed files as a string.

If Emacs was compiled with support for native compilation (see Chapter 18 [Native Compilation], page 328), then when a '`.elc`' byte-compiled file is found by searching `load-path`, Emacs will try to look for a corresponding '`.eln`' file holding the corresponding natively-compiled code. The natively-compiled files are looked up in the directories listed by the `native-comp-eln-load-path`.

`native-comp-eln-load-path` [Variable]
: This variable holds a list of directories where Emacs looks for natively-compiled '`.eln`' files. File names in the list that are not absolute are interpreted as relative to `invocation-directory` (see Section 42.3 [System Environment], page 1270). The last directory in the list is the system directory, i.e. the directory with '`.eln`' files installed by the Emacs build and installation procedure. In each of the directories in the list, Emacs looks for '`.eln`' files in a subdirectory whose name is constructed from the Emacs version and an 8-character hash that depends on the current native-compilation ABI; the name of this subdirectory is stored in the variable `comp-native-version-dir`.

16.4 Loading Non-ASCII Characters

When Emacs Lisp programs contain string constants with non-ASCII characters, these can be represented within Emacs either as unibyte strings or as multibyte strings (see Section 34.1 [Text Representations], page 966). Which representation is used depends on how the file is read into Emacs. If it is read with decoding into multibyte representation, the text of the Lisp program will be multibyte text, and its string constants will be multibyte strings. If a file containing Latin-1 characters (for example) is read without decoding, the text of the program will be unibyte text, and its string constants will be unibyte strings. See Section 34.10 [Coding Systems], page 980.

In most Emacs Lisp programs, the fact that non-ASCII strings are multibyte strings should not be noticeable, since inserting them in unibyte buffers converts them to unibyte automatically. However, if this does make a difference, you can force a particular Lisp file to be interpreted as unibyte by writing '`coding: raw-text`' in a local variables section. With that designator, the file will unconditionally be interpreted as unibyte. This can matter when making key bindings to non-ASCII characters written as `?v`*`literal`*.

16.5 Autoload

The *autoload* facility lets you register the existence of a function or macro, but put off loading the file that defines it. The first call to the function automatically loads the proper library, in order to install the real definition and other associated code, then runs the real definition as if it had been loaded all along. Autoloading can also be triggered by looking up the documentation of the function or macro (see Section 25.1 [Documentation Basics], page 597), and completion of variable and function names (see Section 16.5.1 [Autoload by Prefix], page 308, below).

There are two ways to set up an autoloaded function: by calling `autoload`, and by writing a "magic" comment in the source before the real definition. `autoload` is the low-level primitive for autoloading; any Lisp program can call `autoload` at any time. Magic comments are the most convenient way to make a function autoload, for packages installed along with Emacs. These comments do nothing on their own, but they serve as a guide for the command `loaddefs-generate`, which constructs calls to `autoload` and arranges to execute them when Emacs is built.

`autoload` *function filename* **&optional** *docstring interactive type* [Function]

This function defines the function (or macro) named *function* so as to load automatically from *filename*. The string *filename* specifies the file to load to get the real definition of *function*.

If *filename* does not contain either a directory name, or the suffix `.el` or `.elc`, this function insists on adding one of these suffixes, and it will not load from a file whose name is just *filename* with no added suffix. (The variable `load-suffixes` specifies the exact required suffixes.)

The argument *docstring* is the documentation string for the function. Specifying the documentation string in the call to `autoload` makes it possible to look at the documentation without loading the function's real definition. Normally, this should be identical to the documentation string in the function definition itself. If it isn't, the function definition's documentation string takes effect when it is loaded.

If *interactive* is non-`nil`, that says *function* can be called interactively. This lets completion in `M-x` work without loading *function*'s real definition. The complete interactive specification is not given here; it's not needed unless the user actually calls *function*, and when that happens, it's time to load the real definition.

If *interactive* is a list, it is interpreted as a list of modes this command is applicable for.

You can autoload macros and keymaps as well as ordinary functions. Specify *type* as `macro` if *function* is really a macro. Specify *type* as `keymap` if *function* is really a keymap. Various parts of Emacs need to know this information without loading the real definition.

An autoloaded keymap loads automatically during key lookup when a prefix key's binding is the symbol *function*. Autoloading does not occur for other kinds of access to the keymap. In particular, it does not happen when a Lisp program gets the keymap from the value of a variable and calls `keymap-set`; not even if the variable name is the same symbol *function*.

If *function* already has a non-void function definition that is not an autoload object, this function does nothing and returns `nil`. Otherwise, it constructs an autoload object (see Section 2.4.19 [Autoload Type], page 25), and stores it as the function definition for *function*. The autoload object has this form:

```
(autoload filename docstring interactive type)
```

For example,

```
(symbol-function 'run-prolog)
     ⇒ (autoload "prolog" 169681 t nil)
```

In this case, `"prolog"` is the name of the file to load, 169681 refers to the documentation string in the `emacs/etc/DOC` file (see Section 25.1 [Documentation Basics], page 597), `t` means the function is interactive, and `nil` that it is not a macro or a keymap.

`autoloadp` *object* [Function]

This function returns non-`nil` if *object* is an autoload object. For example, to check if `run-prolog` is defined as an autoloaded function, evaluate

```
(autoloadp (symbol-function 'run-prolog))
```

The autoloaded file usually contains other definitions and may require or provide one or more features. If the file is not completely loaded (due to an error in the evaluation of its contents), any function definitions or `provide` calls that occurred during the load are undone. This is to ensure that the next attempt to call any function autoloading from this file will try again to load the file. If not for this, then some of the functions in the file might be defined by the aborted load, but fail to work properly for the lack of certain subroutines not loaded successfully because they come later in the file.

If the autoloaded file fails to define the desired Lisp function or macro, then an error is signaled with data `"Autoloading failed to define function function-name"`.

A magic autoload comment (often called an *autoload cookie*) consists of '`;;;###autoload`', on a line by itself, just before the real definition of the function in its autoloadable source file. The function `loaddefs-generate` writes a corresponding

`autoload` call into `loaddefs.el`. (The string that serves as the autoload cookie and the name of the file generated by `loaddefs-generate` can be changed from the above defaults, see below.) Building Emacs loads `loaddefs.el` and thus calls `autoload`.

The same magic comment can copy any kind of form into `loaddefs.el`. The form following the magic comment is copied verbatim, *except* if it is one of the forms which the autoload facility handles specially (e.g., by conversion into an `autoload` call). The forms which are not copied verbatim are the following:

Definitions for function or function-like objects:
: `defun` and `defmacro`; also `cl-defun` and `cl-defmacro` (see Section "Argument Lists" in *Common Lisp Extensions*), and `define-overloadable-function` (see the commentary in `mode-local.el`).

Definitions for major or minor modes:
: `define-minor-mode`, `define-globalized-minor-mode`, `define-generic-mode`, `define-derived-mode`, `easy-mmode-define-minor-mode`, `easy-mmode-define-global-mode`, `define-compilation-mode`, and `define-global-minor-mode`.

Other definition types:
: `defcustom`, `defgroup`, `deftheme`, `defclass` (see *EIEIO*), and `define-skeleton` (see *Autotyping*).

You can also use a magic comment to execute a form at build time *without* executing it when the file itself is loaded. To do this, write the form *on the same line* as the magic comment. Since it is in a comment, it does nothing when you load the source file; but `loaddefs-generate` copies it to `loaddefs.el`, where it is executed while building Emacs.

The following example shows how `doctor` is prepared for autoloading with a magic comment:

```
;;;###autoload
(defun doctor ()
  "Switch to *doctor* buffer and start giving psychotherapy."
  (interactive)
  (switch-to-buffer "*doctor*")
  (doctor-mode))
```

Here's what that produces in `loaddefs.el`:

```
(autoload 'doctor "doctor" "\
Switch to *doctor* buffer and start giving psychotherapy.

\(fn)" t nil)
```

While the `loaddefs.el` isn't for editing, we try to keep it somewhat readable for people. For instance, control characters in `defvar` values are escaped, and we insert a backslash and newline immediately following the double-quote of the doc string to keep the line length down. '`(fn)`' in the usage part of the documentation string is replaced with the function's name when the various help functions (see Section 25.6 [Help Functions], page 604) display it.

If you write a function definition with an unusual macro that is not one of the known and recognized function definition methods, use of an ordinary magic autoload comment

would copy the whole definition into `loaddefs.el`. That is not desirable. You can put the desired `autoload` call into `loaddefs.el` instead by writing this:

```
;;;###autoload (autoload 'foo "myfile")
(mydefunmacro foo
  ...)
```

You can use a non-default string as the autoload cookie and have the corresponding autoload calls written into a file whose name is different from the default `loaddefs.el`. Emacs provides two variables to control this:

`lisp-mode-autoload-regexp` [Variable]
: The value of this constant is a regexp that matches autoload cookies. `loaddefs-generate` copies the Lisp form that follows the cookie into the autoload file it generates. This will match comments like '`;;;###autoload`' and '`;;;###calc-autoload`'.

`generated-autoload-file` [Variable]
: The value of this variable names an Emacs Lisp file where the autoload calls should go. The default value is `loaddefs.el`, but you can override that, e.g., in the local variables section of a `.el` file (see Section 12.12 [File Local Variables], page 213). The autoload file is assumed to contain a trailer starting with a formfeed character.

The following function may be used to explicitly load the library specified by an autoload object:

`autoload-do-load` *autoload* **&optional** *name macro-only* [Function]
: This function performs the loading specified by *autoload*, which should be an autoload object. The optional argument *name*, if non-`nil`, should be a symbol whose function value is *autoload*; in that case, the return value of this function is the symbol's new function value. If the value of the optional argument *macro-only* is `macro`, this function avoids loading a function, only a macro.

16.5.1 Autoload by Prefix

During completion for the commands `describe-variable` and `describe-function`, Emacs will try to load files which may contain definitions matching the prefix being completed. The variable `definition-prefixes` holds a hashtable which maps a prefix to the corresponding list of files to load for it. Entries to this mapping are added by calls to `register-definition-prefixes` which are generated by `loaddefs-generate` (see Section 16.5 [Autoload], page 305). Files which don't contain any definitions worth loading (test files, for example), should set `autoload-compute-prefixes` to `nil` as a file-local variable.

16.5.2 When to Use Autoload

Do not add an autoload comment unless it is really necessary. Autoloading code means it is always globally visible. Once an item is autoloaded, there is no compatible way to transition back to it not being autoloaded (after people become accustomed to being able to use it without an explicit load).

- The most common items to autoload are the interactive entry points to a library. For example, if `python.el` is a library defining a major-mode for editing Python code,

autoload the definition of the `python-mode` function, so that people can simply use *`M-x python-mode`* to load the library.

- Variables usually don't need to be autoloaded. An exception is if the variable on its own is generally useful without the whole defining library being loaded. (An example of this might be something like `find-exec-terminator`.)
- Don't autoload a user option just so that a user can set it.
- Never add an autoload *comment* to silence a compiler warning in another file. In the file that produces the warning, use `(defvar foo)` to silence an undefined variable warning, and `declare-function` (see Section 13.16 [Declaring Functions], page 265) to silence an undefined function warning; or require the relevant library; or use an explicit autoload *statement*.

16.6 Repeated Loading

You can load a given file more than once in an Emacs session. For example, after you have rewritten and reinstalled a function definition by editing it in a buffer, you may wish to return to the original version; you can do this by reloading the file it came from.

When you load or reload files, bear in mind that the `load` and `load-library` functions automatically load a byte-compiled file rather than a non-compiled file of similar name. If you rewrite a file that you intend to save and reinstall, you need to byte-compile the new version; otherwise Emacs will load the older, byte-compiled file instead of your newer, non-compiled file! If that happens, the message displayed when loading the file includes, '`(compiled; note, source is newer)`', to remind you to recompile it.

When writing the forms in a Lisp library file, keep in mind that the file might be loaded more than once. For example, think about whether each variable should be reinitialized when you reload the library; `defvar` does not change the value if the variable is already initialized. (See Section 12.5 [Defining Variables], page 194.)

The simplest way to add an element to an alist is like this:

```
(push '(leif-mode " Leif") minor-mode-alist)
```

But this would add multiple elements if the library is reloaded. To avoid the problem, use `add-to-list` (see Section 5.5 [List Variables], page 85):

```
(add-to-list 'minor-mode-alist '(leif-mode " Leif"))
```

Occasionally you will want to test explicitly whether a library has already been loaded. If the library uses `provide` to provide a named feature, you can use `featurep` earlier in the file to test whether the `provide` call has been executed before (see Section 16.7 [Named Features], page 310). Alternatively, you could use something like this:

```
(defvar foo-was-loaded nil)

(unless foo-was-loaded
  execute-first-time-only
  (setq foo-was-loaded t))
```

16.7 Features

`provide` and `require` are an alternative to `autoload` for loading files automatically. They work in terms of named *features*. Autoloading is triggered by calling a specific function, but a feature is loaded the first time another program asks for it by name.

A feature name is a symbol that stands for a collection of functions, variables, etc. The file that defines them should *provide* the feature. Another program that uses them may ensure they are defined by *requiring* the feature. This loads the file of definitions if it hasn't been loaded already.

To require the presence of a feature, call `require` with the feature name as argument. `require` looks in the global variable `features` to see whether the desired feature has been provided already. If not, it loads the feature from the appropriate file. This file should call `provide` at the top level to add the feature to `features`; if it fails to do so, `require` signals an error.

For example, in `idlwave.el`, the definition for `idlwave-complete-filename` includes the following code:

```
(defun idlwave-complete-filename ()
  "Use the comint stuff to complete a file name."
   (require 'comint)
   (let* ((comint-file-name-chars "~/A-Za-z0-9+@:_.$#%={}\\-")
          (comint-completion-addsuffix nil)
          ...)
       (comint-dynamic-complete-filename)))
```

The expression `(require 'comint)` loads the file `comint.el` if it has not yet been loaded, ensuring that `comint-dynamic-complete-filename` is defined. Features are normally named after the files that provide them, so that `require` need not be given the file name. (Note that it is important that the `require` statement be outside the body of the `let`. Loading a library while its variables are let-bound can have unintended consequences, namely the variables becoming unbound after the let exits.)

The `comint.el` file contains the following top-level expression:

```
(provide 'comint)
```

This adds `comint` to the global `features` list, so that `(require 'comint)` will henceforth know that nothing needs to be done.

When `require` is used at top level in a file, it takes effect when you byte-compile that file (see Chapter 17 [Byte Compilation], page 317) as well as when you load it. This is in case the required package contains macros that the byte compiler must know about. It also avoids byte compiler warnings for functions and variables defined in the file loaded with `require`.

Although top-level calls to `require` are evaluated during byte compilation, `provide` calls are not. Therefore, you can ensure that a file of definitions is loaded before it is byte-compiled by including a `provide` followed by a `require` for the same feature, as in the following example.

```
(provide 'my-feature)  ; Ignored by byte compiler,
                       ;    evaluated by load.
(require 'my-feature)  ; Evaluated by byte compiler.
```

The compiler ignores the `provide`, then processes the `require` by loading the file in question. Loading the file does execute the `provide` call, so the subsequent `require` call does nothing when the file is loaded.

`provide` *feature* **&optional** *subfeatures* [Function]

This function announces that *feature* is now loaded, or being loaded, into the current Emacs session. This means that the facilities associated with *feature* are or will be available for other Lisp programs.

The direct effect of calling `provide` is to add *feature* to the front of `features` if it is not already in that list and call any `eval-after-load` code waiting for it (see Section 16.10 [Hooks for Loading], page 314). The argument *feature* must be a symbol. `provide` returns *feature*.

If provided, *subfeatures* should be a list of symbols indicating a set of specific subfeatures provided by this version of *feature*. You can test the presence of a subfeature using `featurep`. The idea of subfeatures is that you use them when a package (which is one *feature*) is complex enough to make it useful to give names to various parts or functionalities of the package, which might or might not be loaded, or might or might not be present in a given version. See Section 40.17.3 [Network Feature Testing], page 1122, for an example.

```
features
     ⇒ (bar bish)

(provide 'foo)
     ⇒ foo
features
     ⇒ (foo bar bish)
```

When a file is loaded to satisfy an autoload, and it stops due to an error in the evaluation of its contents, any function definitions or `provide` calls that occurred during the load are undone. See Section 16.5 [Autoload], page 305.

`require` *feature* **&optional** *filename noerror* [Function]

This function checks whether *feature* is present in the current Emacs session (using `(featurep` *`feature`*`)`; see below). The argument *feature* must be a symbol.

If the feature is not present, then `require` loads *filename* with `load`. If *filename* is not supplied, then the name of the symbol *feature* is used as the base file name to load. However, in this case, `require` insists on finding *feature* with an added '`.el`' or '`.elc`' suffix (possibly extended with a compression suffix); a file whose name is just *feature* won't be used. (The variable `load-suffixes` specifies the exact required Lisp suffixes.)

If *noerror* is non-`nil`, that suppresses errors from actual loading of the file. In that case, `require` returns `nil` if loading the file fails. Normally, `require` returns *feature*.

If loading the file succeeds but does not provide *feature*, `require` signals an error about the missing feature.

`featurep` *feature* **&optional** *subfeature* [Function]

This function returns `t` if *feature* has been provided in the current Emacs session (i.e., if *feature* is a member of `features`.) If *subfeature* is non-`nil`, then the function

returns `t` only if that subfeature is provided as well (i.e., if *subfeature* is a member of the `subfeature` property of the *feature* symbol.)

`features` [Variable]
: The value of this variable is a list of symbols that are the features loaded in the current Emacs session. Each symbol was put in this list with a call to `provide`. The order of the elements in the `features` list is not significant.

The `use-package` macro provides a convenient way of loading a feature and configuring it for use. It provides a means to combine requiring a feature, like `require` does, with code to be run when the feature is actually loaded, similar to load-time hooks (see Section 16.10 [Hooks for Loading], page 314). The declarative syntax of `use-package` makes it exceptionally easy to use in user init files.

`use-package` *feature* **&rest** *args* [Macro]
: This macro specifies how to load the named *feature* and how to configure and customize it for use. The arguments *args* are keyword-value pairs. Some of the important keywords and their values are:

  `:init` *forms*
  : Specifies *forms* to execute before *feature* is loaded.

  `:config` *forms*
  : Specifies *forms* to execute after loading *feature*.

  `:defer` *condition*
  : If *condition* is non-`nil`, it specifies to defer loading *feature* until any of the autoloaded commands or variables of *feature* are first used. If *condition* is a number *n*, it specifies that *feature* should be loaded after *n* seconds of idle time.

  `:commands` *commands...*
  : Specifies commands of *feature* to be autoloaded.

  `:bind` *keybindings...*
  : Specifies the *keybindings* for *feature*s commands. Each binding has the form

    ```
    (key-sequence . command)
    ```

    or

    ```
    (:map keymap (key-sequence . command))
    ```

    where *key-sequence* is in the form accepted by the `kbd` macro (see Section 23.1 [Key Sequences], page 481).

  For more details about `use-package`, see *use-package User Manual*.

16.8 Which File Defined a Certain Symbol

`symbol-file` *symbol* **&optional** *type native-p* [Function]
: This function returns the name of the file that defined *symbol*. If *type* is `nil`, then any kind of definition is acceptable. If *type* is `defun`, `defvar`, or `defface`, that specifies function definition, variable definition, or face definition only.

The value is normally an absolute file name. It can also be `nil`, if the definition is not associated with any file. If *symbol* specifies an autoloaded function, the value can be a relative file name without extension.

If the optional third argument *native-p* is non-`nil`, and Emacs was built with native compilation support (see Chapter 18 [Native Compilation], page 328), this function will try to find the `.eln` file that defined *symbol*, instead of the `.elc` or `.el` file. If such a `.eln` file is found and is not outdated, the function will return its absolute file name; otherwise it will report the name of either the source or the byte-compiled file.

The basis for `symbol-file` is the data in the variable `load-history`.

`load-history` [Variable]

The value of this variable is an alist that associates the names of loaded library files with the names of the functions and variables they defined, as well as the features they provided or required.

Each element in this alist describes one loaded library (including libraries that are preloaded at startup). It is a list whose CAR is the absolute file name of the library (a string). The rest of the list elements have these forms:

var The symbol *var* was defined as a variable.

`(defun . `*fun*`)`
The function *fun* was defined. `(defun . `*fun*`)`, which represents defining *fun* as a function.

`(defface . `*face*`)`
The face *face* was defined.

`(require . `*feature*`)`
The feature *feature* was required.

`(provide . `*feature*`)`
The feature *feature* was provided.

`(cl-defmethod `*method specializers*`)`
The named *method* was defined by using `cl-defmethod`, with *specializers* as its specializers.

`(define-type . `*type*`)`
The type *type* was defined.

The value of `load-history` may have one element whose CAR is `nil`. This element describes definitions made with `eval-buffer` on a buffer that is not visiting a file.

The command `eval-region` updates `load-history`, but does so by adding the symbols defined to the element for the file being visited, rather than replacing that element. See Section 10.4 [Eval], page 152.

In addition to `load-history`, every function keeps track of its own history in the symbol property `function-history`. The reason why functions are treated specially in this respect is that it is common for functions to be defined in two steps in two different files (typically, one of them is an autoload), so in order to be able to properly *unload* a file, we need to know more precisely what that file did to the function definition.

The symbol property `function-history` holds a list of the form `(`*`file1 def2 file2 def3 ...`*`)`, where *file1* is the last file that changed the definition and *def2* was the definition before *file1*, set by *file2*, etc. Logically this list should end with the name of the first file that defined this function, but to save space this last element is usually omitted.

16.9 Unloading

You can discard the functions and variables loaded by a library to reclaim memory for other Lisp objects. To do this, use the function `unload-feature`:

`unload-feature` *feature* **&optional** *force* [Command]

This command unloads the library that provided feature *feature*. It undefines all functions, macros, and variables defined in that library with `defun`, `defalias`, `defsubst`, `defmacro`, `defconst`, `defvar`, and `defcustom`. It then restores any autoloads formerly associated with those symbols. (Loading saves these in the `function-history` property of the symbol.)

Before restoring the previous definitions, `unload-feature` runs `remove-hook` to remove functions defined by the library from certain hooks. These hooks include variables whose names end in '`-hook`' (or the deprecated suffix '`-hooks`'), plus those listed in `unload-feature-special-hooks`, as well as `auto-mode-alist`. This is to prevent Emacs from ceasing to function because important hooks refer to functions that are no longer defined.

Standard unloading activities also undo ELP profiling of functions in that library, unprovides any features provided by the library, and cancels timers held in variables defined by the library.

If these measures are not sufficient to prevent malfunction, a library can define an explicit unloader named *`feature`*`-unload-function`. If that symbol is defined as a function, `unload-feature` calls it with no arguments before doing anything else. It can do whatever is appropriate to unload the library. If it returns `nil`, `unload-feature` proceeds to take the normal unload actions. Otherwise it considers the job to be done.

Ordinarily, `unload-feature` refuses to unload a library on which other loaded libraries depend. (A library *a* depends on library *b* if *a* contains a `require` for *b*.) If the optional argument *force* is non-`nil`, dependencies are ignored and you can unload any library.

The `unload-feature` function is written in Lisp; its actions are based on the variable `load-history`.

`unload-feature-special-hooks` [Variable]

This variable holds a list of hooks to be scanned before unloading a library, to remove functions defined in the library.

16.10 Hooks for Loading

You can ask for code to be executed each time Emacs loads a library, by using the variable `after-load-functions`:

`after-load-functions` [Variable]
: This abnormal hook is run after loading a file. Each function in the hook is called with a single argument, the absolute filename of the file that was just loaded.

If you want code to be executed when a *particular* library is loaded, use the macro `with-eval-after-load`:

`with-eval-after-load` *library body...* [Macro]
: This macro arranges to evaluate *body* at the end of loading the file *library*, each time *library* is loaded. If *library* is already loaded, it evaluates *body* right away.

  You don't need to give a directory or extension in the file name *library*. Normally, you just give a bare file name, like this:

```
(with-eval-after-load "js" (keymap-set js-mode-map "C-c C-c" 'js-eval))
```

  To restrict which files can trigger the evaluation, include a directory or an extension or both in *library*. Only a file whose absolute true name (i.e., the name with all symbolic links chased out) matches all the given name components will match. In the following example, `my_inst.elc` or `my_inst.elc.gz` in some directory `..../foo/bar` will trigger the evaluation, but not `my_inst.el`:

```
(with-eval-after-load "foo/bar/my_inst.elc" ...)
```

  library can also be a feature (i.e., a symbol), in which case *body* is evaluated at the end of any file where `(provide` *`library`*`)` is called.

  An error in *body* does not undo the load, but does prevent execution of the rest of *body*.

Normally, well-designed Lisp programs should not use `with-eval-after-load`. If you need to examine and set the variables defined in another library (those meant for outside use), you can do it immediately—there is no need to wait until the library is loaded. If you need to call functions defined by that library, you should load the library, preferably with `require` (see Section 16.7 [Named Features], page 310).

16.11 Emacs Dynamic Modules

A *dynamic Emacs module* is a shared library that provides additional functionality for use in Emacs Lisp programs, just like a package written in Emacs Lisp would.

Functions that load Emacs Lisp packages can also load dynamic modules. They recognize dynamic modules by looking at their file-name extension, a.k.a. "suffix". This suffix is platform-dependent.

`module-file-suffix` [Variable]
: This variable holds the system-dependent value of the file-name extension of the module files. Its value is `.so` on POSIX hosts, `.dylib` on macOS, and `.dll` on MS-Windows.

On macOS, dynamic modules can also have the suffix `.so` in addition to `.dylib`.

Every dynamic module should export a C-callable function named `emacs_module_init`, which Emacs will call as part of the call to `load` or `require` which loads the module. It should also export a symbol named `plugin_is_GPL_compatible` to indicate that its code

is released under the GPL or compatible license; Emacs will signal an error if your program tries to load modules that don't export such a symbol.

If a module needs to call Emacs functions, it should do so through the API (Application Programming Interface) defined and documented in the header file `emacs-module.h` that is part of the Emacs distribution. See Section E.8 [Writing Dynamic Modules], page 1366, for details of using that API when writing your own modules.

Modules can create `user-ptr` Lisp objects that embed pointers to C struct's defined by the module. This is useful for keeping around complex data structures created by a module, to be passed back to the module's functions. User-ptr objects can also have associated *finalizers* – functions to be run when the object is GC'ed; this is useful for freeing any resources allocated for the underlying data structure, such as memory, open file descriptors, etc. See Section E.8.3 [Module Values], page 1371.

`user-ptrp` *object* [Function]
: This function returns `t` if its argument is a `user-ptr` object.

`module-load` *file* [Function]
: Emacs calls this low-level primitive to load a module from the specified *file* and perform the necessary initialization of the module. This is the primitive which makes sure the module exports the `plugin_is_GPL_compatible` symbol, calls the module's `emacs_module_init` function, and signals an error if that function returns an error indication, or if the user typed *C-g* during the initialization. If the initialization succeeds, `module-load` returns `t`. Note that *file* must already have the proper file-name extension, as this function doesn't try looking for files with known extensions, unlike `load`.

  Unlike `load`, `module-load` doesn't record the module in `load-history`, doesn't print any messages, and doesn't protect against recursive loads. Most users should therefore use `load`, `load-file`, `load-library`, or `require` instead of `module-load`.

Loadable modules in Emacs are enabled by using the *--with-modules* option at configure time.

17 Byte Compilation

Emacs Lisp has a *compiler* that translates functions written in Lisp into a special representation called *byte-code* that can be executed more efficiently. The compiler replaces Lisp function definitions with byte-code. When a byte-code function is called, its definition is evaluated by the *byte-code interpreter*.

Because the byte-compiled code is evaluated by the byte-code interpreter, instead of being executed directly by the machine's hardware (as true compiled code is), byte-code is completely transportable from machine to machine without recompilation. It is not, however, as fast as true compiled code.

In general, any version of Emacs can run byte-compiled code produced by recent earlier versions of Emacs, but the reverse is not true.

If you do not want a Lisp file to be compiled, ever, put a file-local variable binding for `no-byte-compile` into it, like this:

```
;; -*-no-byte-compile: t; -*-
```

17.1 Performance of Byte-Compiled Code

A byte-compiled function is not as efficient as a primitive function written in C, but runs much faster than the version written in Lisp. Here is an example:

```
(defun silly-loop (n)
  "Return the time, in seconds, to run N iterations of a loop."
  (let ((t1 (float-time)))
    (while (> (setq n (1- n)) 0))
    (- (float-time) t1)))
⇒ silly-loop

(silly-loop 50000000)
⇒ 5.200886011123657

(byte-compile 'silly-loop)
⇒ [Compiled code not shown]

(silly-loop 50000000)
⇒ 0.6239290237426758
```

In this example, the interpreted code required more than 5 seconds to run, whereas the byte-compiled code required less than 1 second. These results are representative, but actual results may vary.

17.2 Byte-Compilation Functions

You can byte-compile an individual function or macro definition with the `byte-compile` function. You can compile a whole file with `byte-compile-file`, or several files with `byte-recompile-directory` or `batch-byte-compile`.

Sometimes, the byte compiler produces warning and/or error messages (see Section 17.6 [Compiler Errors], page 322, for details). These messages are normally recorded in a buffer

called `*Compile-Log*`, which uses Compilation mode. See Section "Compilation Mode" in *The GNU Emacs Manual*. However, if the variable `byte-compile-debug` is non-`nil`, error messages will be signaled as Lisp errors instead (see Section 11.7.3 [Errors], page 179).

Be careful when writing macro calls in files that you intend to byte-compile. Since macro calls are expanded when they are compiled, the macros need to be loaded into Emacs or the byte compiler will not do the right thing. The usual way to handle this is with `require` forms which specify the files containing the needed macro definitions (see Section 16.7 [Named Features], page 310). Normally, the byte compiler does not evaluate the code that it is compiling, but it handles `require` forms specially, by loading the specified libraries. To avoid loading the macro definition files when someone *runs* the compiled program, write `eval-when-compile` around the `require` calls (see Section 17.5 [Eval During Compile], page 321). For more details, See Section 14.3 [Compiling Macros], page 271.

Inline (`defsubst`) functions are less troublesome; if you compile a call to such a function before its definition is known, the call will still work right, it will just run slower.

`byte-compile` *symbol* [Function]

This function byte-compiles the function definition of *symbol*, replacing the previous definition with the compiled one. The function definition of *symbol* must be the actual code for the function; `byte-compile` does not handle function indirection. The return value is the byte-code function object which is the compiled definition of *symbol* (see Section 17.7 [Byte-Code Objects], page 324).

```
(defun factorial (integer)
  "Compute factorial of INTEGER."
  (if (= 1 integer) 1
    (* integer (factorial (1- integer)))))
⇒ factorial

(byte-compile 'factorial)
⇒
#[257
  "\211\300U\203^H^@\300\207\211\301^BS!_\207"
  [1 factorial] 4
  "Compute factorial of INTEGER.\n\n(fn INTEGER)"]
```

If *symbol*'s definition is a byte-code function object, `byte-compile` does nothing and returns `nil`. It does not compile the symbol's definition again, since the original (non-compiled) code has already been replaced in the symbol's function cell by the byte-compiled code.

The argument to `byte-compile` can also be a `lambda` expression. In that case, the function returns the corresponding compiled code but does not store it anywhere.

`compile-defun` **&optional** *arg* [Command]

This command reads the defun containing point, compiles it, and evaluates the result. If you use this on a defun that is actually a function definition, the effect is to install a compiled version of that function.

`compile-defun` normally displays the result of evaluation in the echo area, but if *arg* is non-`nil`, it inserts the result in the current buffer after the form it has compiled.

`byte-compile-file` *filename* [Command]

This function compiles a file of Lisp code named *filename* into a file of byte-code. The output file's name is made by changing the '`.el`' suffix into '`.elc`'; if *filename* does not end in '`.el`', it adds '`.elc`' to the end of *filename*.

Compilation works by reading the input file one form at a time. If it is a definition of a function or macro, the compiled function or macro definition is written out. Other forms are batched together, then each batch is compiled, and written so that its compiled code will be executed when the file is read. All comments are discarded when the input file is read.

This command returns `t` if there were no errors and `nil` otherwise. When called interactively, it prompts for the file name.

```
$ ls -l push*
-rw-r--r-- 1 lewis lewis 791 Oct  5 20:31 push.el

(byte-compile-file "~/emacs/push.el")
     ⇒ t

$ ls -l push*
-rw-r--r-- 1 lewis lewis 791 Oct  5 20:31 push.el
-rw-rw-rw- 1 lewis lewis 638 Oct  8 20:25 push.elc
```

`byte-recompile-directory` *directory* **&optional** *flag force follow-symlinks* [Command]

This command recompiles every '`.el`' file in *directory* (or its subdirectories) that needs recompilation. A file needs recompilation if a '`.elc`' file exists but is older than the '`.el`' file.

When a '`.el`' file has no corresponding '`.elc`' file, *flag* says what to do. If it is `nil`, this command ignores these files. If *flag* is 0, it compiles them. If it is neither `nil` nor 0, it asks the user whether to compile each such file, and asks about each subdirectory as well.

Interactively, `byte-recompile-directory` prompts for *directory* and *flag* is the prefix argument.

If *force* is non-`nil`, this command recompiles every '`.el`' file that has a '`.elc`' file.

This command will normally not compile '`.el`' files that are symlinked. If the optional *follow-symlink* parameter is non-`nil`, symlinked '`.el`' will also be compiled.

The returned value is unpredictable.

`batch-byte-compile` **&optional** *noforce* [Function]

This function runs `byte-compile-file` on files specified on the command line. This function must be used only in a batch execution of Emacs, as it kills Emacs on completion. An error in one file does not prevent processing of subsequent files, but no output file will be generated for it, and the Emacs process will terminate with a nonzero status code.

If *noforce* is non-`nil`, this function does not recompile files that have an up-to-date '`.elc`' file.

```
$ emacs -batch -f batch-byte-compile *.el
```

17.3 Documentation Strings and Compilation

When Emacs loads functions and variables from a byte-compiled file, it normally does not load their documentation strings into memory. Each documentation string is dynamically loaded from the byte-compiled file only when needed. This saves memory, and speeds up loading by skipping the processing of the documentation strings.

This feature has a drawback: if you delete, move, or alter the compiled file (such as by compiling a new version), Emacs may no longer be able to access the documentation string of previously-loaded functions or variables. Such a problem normally only occurs if you build Emacs yourself, and happen to edit and/or recompile the Lisp source files. To solve it, just reload each file after recompilation.

Dynamic loading of documentation strings from byte-compiled files is determined, at compile time, for each byte-compiled file. It can be disabled via the option `byte-compile-dynamic-docstrings`.

`byte-compile-dynamic-docstrings` [User Option]

If this is non-`nil`, the byte compiler generates compiled files that are set up for dynamic loading of documentation strings.

To disable the dynamic loading feature for a specific file, set this option to `nil` in its header line (see Section "Local Variables in Files" in *The GNU Emacs Manual*), like this:

```
-*-byte-compile-dynamic-docstrings: nil;-*-
```

This is useful mainly if you expect to change the file, and you want Emacs sessions that have already loaded it to keep working when the file changes.

Internally, the dynamic loading of documentation strings is accomplished by writing compiled files with a special Lisp reader construct, '`#@`*count*'. This construct skips the next *count* characters. It also uses the '`#$`' construct, which stands for the name of this file, as a string. Do not use these constructs in Lisp source files; they are not designed to be clear to humans reading the file.

17.4 Dynamic Loading of Individual Functions

When you compile a file, you can optionally enable the *dynamic function loading* feature (also known as *lazy loading*). With dynamic function loading, loading the file doesn't fully read the function definitions in the file. Instead, each function definition contains a place-holder which refers to the file. The first time each function is called, it reads the full definition from the file, to replace the place-holder.

The advantage of dynamic function loading is that loading the file should become faster. This is a good thing for a file which contains many separate user-callable functions, if using one of them does not imply you will probably also use the rest. A specialized mode which provides many keyboard commands often has that usage pattern: a user may invoke the mode, but use only a few of the commands it provides.

The dynamic loading feature has certain disadvantages:

- If you delete or move the compiled file after loading it, Emacs can no longer load the remaining function definitions not already loaded.

- If you alter the compiled file (such as by compiling a new version), then trying to load any function not already loaded will usually yield nonsense results.

These problems will never happen in normal circumstances with installed Emacs files. But they are quite likely to happen with Lisp files that you are changing. The easiest way to prevent these problems is to reload the new compiled file immediately after each recompilation.

Experience shows that using dynamic function loading provides benefits that are hardly measurable, so this feature is deprecated since Emacs 27.1.

The byte compiler uses the dynamic function loading feature if the variable `byte-compile-dynamic` is non-`nil` at compilation time. Do not set this variable globally, since dynamic loading is desirable only for certain files. Instead, enable the feature for specific source files with file-local variable bindings. For example, you could do it by writing this text in the source file's first line:

```
-*-byte-compile-dynamic: t;-*-
```

`byte-compile-dynamic` [Variable]
: If this is non-`nil`, the byte compiler generates compiled files that are set up for dynamic function loading.

`fetch-bytecode` *function* [Function]
: If *function* is a byte-code function object, this immediately finishes loading the byte code of *function* from its byte-compiled file, if it is not fully loaded already. Otherwise, it does nothing. It always returns *function*.

17.5 Evaluation During Compilation

These features permit you to write code to be evaluated during compilation of a program.

`eval-and-compile` *body...* [Macro]
: This form marks *body* to be evaluated both when you compile the containing code and when you run it (whether compiled or not).

  You can get a similar result by putting *body* in a separate file and referring to that file with `require`. That method is preferable when *body* is large. Effectively `require` is automatically `eval-and-compile`, the package is loaded both when compiling and executing.

  `autoload` is also effectively `eval-and-compile` too. It's recognized when compiling, so uses of such a function don't produce "not known to be defined" warnings.

  Most uses of `eval-and-compile` are fairly sophisticated.

  If a macro has a helper function to build its result, and that macro is used both locally and outside the package, then `eval-and-compile` should be used to get the helper both when compiling and then later when running.

  If functions are defined programmatically (with `fset` say), then `eval-and-compile` can be used to have that done at compile-time as well as run-time, so calls to those functions are checked (and warnings about "not known to be defined" suppressed).

`eval-when-compile` *body...* [Macro]

This form marks *body* to be evaluated at compile time but not when the compiled program is loaded. The result of evaluation by the compiler becomes a constant which appears in the compiled program. If you load the source file, rather than compiling it, *body* is evaluated normally.

If you have a constant that needs some calculation to produce, `eval-when-compile` can do that at compile-time. For example,

```
(defvar my-regexp
  (eval-when-compile (regexp-opt '("aaa" "aba" "abb"))))
```

If you're using another package, but only need macros from it (the byte compiler will expand those), then `eval-when-compile` can be used to load it for compiling, but not executing. For example,

```
(eval-when-compile
  (require 'my-macro-package))
```

The same sort of thing goes for macros and `defsubst` functions defined locally and only for use within the file. They are needed for compiling the file, but in most cases they are not needed for execution of the compiled file. For example,

```
(eval-when-compile
  (unless (fboundp 'some-new-thing)
    (defmacro some-new-thing ()
      (compatibility code))))
```

This is often good for code that's only a fallback for compatibility with other versions of Emacs.

Common Lisp Note: At top level, `eval-when-compile` is analogous to the Common Lisp idiom `(eval-when (compile eval) ...)`. Elsewhere, the Common Lisp '`#.`' reader macro (but not when interpreting) is closer to what `eval-when-compile` does.

17.6 Compiler Errors

Error and warning messages from byte compilation are printed in a buffer named `*Compile-Log*`. These messages include file names and line numbers identifying the location of the problem. The usual Emacs commands for operating on compiler output can be used on these messages.

When an error is due to invalid syntax in the program, the byte compiler might get confused about the error's exact location. One way to investigate is to switch to the buffer `*Compiler Input*`. (This buffer name starts with a space, so it does not show up in the Buffer Menu.) This buffer contains the program being compiled, and point shows how far the byte compiler was able to read; the cause of the error might be nearby. See Section 19.3 [Syntax Errors], page 368, for some tips for locating syntax errors.

A common type of warning issued by the byte compiler is for functions and variables that were used but not defined. Such warnings report the line number for the end of the file, not the locations where the missing functions or variables were used; to find these, you must search the file manually.

If you are sure that a warning message about a missing function or variable is unjustified, there are several ways to suppress it:

- You can suppress the warning for a specific call to a function *func* by conditionalizing it on an `fboundp` test, like this:

  ```
  (if (fboundp 'func) ...(func ...)...)
  ```

  The call to *func* must be in the *then-form* of the `if`, and *func* must appear quoted in the call to `fboundp`. (This feature operates for `cond` as well.)

- Likewise, you can suppress the warning for a specific use of a variable *variable* by conditionalizing it on a `boundp` test:

  ```
  (if (boundp 'variable) ...variable...)
  ```

  The reference to *variable* must be in the *then-form* of the `if`, and *variable* must appear quoted in the call to `boundp`.

- You can tell the compiler that a function is defined using `declare-function`. See Section 13.16 [Declaring Functions], page 265.

- Likewise, you can tell the compiler that a variable is defined using `defvar` with no initial value. (Note that this marks the variable as special, i.e. dynamically bound, but only within the current lexical scope, or file if at top-level.) See Section 12.5 [Defining Variables], page 194.

You can also suppress compiler warnings within a certain expression using the `with-suppressed-warnings` macro:

`with-suppressed-warnings` *warnings body...* [Special Form]

In execution, this is equivalent to `(progn body...)`, but the compiler does not issue warnings for the specified conditions in *body*. *warnings* is an association list of warning symbols and function/variable symbols they apply to. For instance, if you wish to call an obsolete function called `foo`, but want to suppress the compilation warning, say:

```
(with-suppressed-warnings ((obsolete foo))
  (foo ...))
```

For more coarse-grained suppression of compiler warnings, you can use the `with-no-warnings` construct:

`with-no-warnings` *body...* [Special Form]

In execution, this is equivalent to `(progn body...)`, but the compiler does not issue warnings for anything that occurs inside *body*.

We recommend that you use `with-suppressed-warnings` instead, but if you do use this construct, that you use it around the smallest possible piece of code to avoid missing possible warnings other than one you intend to suppress.

Byte compiler warnings can be controlled more precisely by setting the variable `byte-compile-warnings`. See its documentation string for details.

Sometimes you may wish the byte-compiler warnings to be reported using `error`. If so, set `byte-compile-error-on-warn` to a non-`nil` value.

17.7 Byte-Code Function Objects

Byte-compiled functions have a special data type: they are *byte-code function objects*. Whenever such an object appears as a function to be called, Emacs uses the byte-code interpreter to execute the byte-code.

Internally, a byte-code function object is much like a vector; its elements can be accessed using `aref`. Its printed representation is like that for a vector, with an additional '`#`' before the opening '`[`'. It must have at least four elements; there is no maximum number, but only the first six elements have any normal use. They are:

argdesc The descriptor of the arguments. This can either be a list of arguments, as described in Section 13.2.3 [Argument List], page 234, or an integer encoding the required number of arguments. In the latter case, the value of the descriptor specifies the minimum number of arguments in the bits zero to 6, and the maximum number of arguments in bits 8 to 14. If the argument list uses `&rest`, then bit 7 is set; otherwise it's cleared.

If *argdesc* is a list, the arguments will be dynamically bound before executing the byte code. If *argdesc* is an integer, the arguments will be instead pushed onto the stack of the byte-code interpreter, before executing the code.

byte-code The string containing the byte-code instructions.

constants The vector of Lisp objects referenced by the byte code. These include symbols used as function names and variable names.

stacksize The maximum stack size this function needs.

docstring The documentation string (if any); otherwise, `nil`. The value may be a number or a list, in case the documentation string is stored in a file. Use the function `documentation` to get the real documentation string (see Section 25.2 [Accessing Documentation], page 598).

interactive
The interactive spec (if any). This can be a string or a Lisp expression. It is `nil` for a function that isn't interactive.

Here's an example of a byte-code function object, in printed representation. It is the definition of the command `backward-sexp`.

```
#[256
  "\211\204^G^@\300\262^A\301^A[!\207"
  [1 forward-sexp]
  3
  1793299
  "^p"]
```

The primitive way to create a byte-code object is with `make-byte-code`:

`make-byte-code` **&rest** *elements* [Function]
This function constructs and returns a byte-code function object with *elements* as its elements.

You should not try to come up with the elements for a byte-code function yourself, because if they are inconsistent, Emacs may crash when you call the function. Always leave it to the byte compiler to create these objects; it makes the elements consistent (we hope).

17.8 Disassembled Byte-Code

People do not write byte-code; that job is left to the byte compiler. But we provide a disassembler to satisfy a cat-like curiosity. The disassembler converts the byte-compiled code into human-readable form.

The byte-code interpreter is implemented as a simple stack machine. It pushes values onto a stack of its own, then pops them off to use them in calculations whose results are themselves pushed back on the stack. When a byte-code function returns, it pops a value off the stack and returns it as the value of the function.

In addition to the stack, byte-code functions can use, bind, and set ordinary Lisp variables, by transferring values between variables and the stack.

`disassemble` *object* **&optional** *buffer-or-name* [Command]

This command displays the disassembled code for *object*. In interactive use, or if *buffer-or-name* is `nil` or omitted, the output goes in a buffer named `*Disassemble*`. If *buffer-or-name* is non-`nil`, it must be a buffer or the name of an existing buffer. Then the output goes there, at point, and point is left before the output.

The argument *object* can be a function name, a lambda expression (see Section 13.2 [Lambda Expressions], page 232), or a byte-code object (see Section 17.7 [Byte-Code Objects], page 324). If it is a lambda expression, `disassemble` compiles it and disassembles the resulting compiled code.

Here are two examples of using the `disassemble` function. We have added explanatory comments to help you relate the byte-code to the Lisp source; these do not appear in the output of `disassemble`.

```
(defun factorial (integer)
  "Compute factorial of an integer."
  (if (= 1 integer) 1
    (* integer (factorial (1- integer)))))
     ⇒ factorial

(factorial 4)
     ⇒ 24

(disassemble 'factorial)
     ⊣ byte-code for factorial:
 doc: Compute factorial of an integer.
 args: (arg1)

0   dup                 ; Get the value of integer and
                        ;   push it onto the stack.
1   constant 1          ; Push 1 onto stack.
2   eqlsign             ; Pop top two values off stack, compare
                        ;   them, and push result onto stack.
```

```
3   goto-if-nil 1          ; Pop and test top of stack;
                           ;    if nil, go to 1, else continue.
6   constant 1             ; Push 1 onto top of stack.
7   return                 ; Return the top element of the stack.
8:1 dup                    ; Push value of integer onto stack.
9   constant factorial     ; Push factorial onto stack.
10  stack-ref 2            ; Push value of integer onto stack.
11  sub1                   ; Pop integer, decrement value,
                           ;    push new value onto stack.
12  call     1             ; Call function factorial using first
                           ;    (i.e., top) stack element as argument;
                           ;    push returned value onto stack.
13  mult                   ; Pop top two values off stack, multiply
                           ;    them, and push result onto stack.
14  return                 ; Return the top element of the stack.
```

The `silly-loop` function is somewhat more complex:

```
(defun silly-loop (n)
  "Return time before and after N iterations of a loop."
  (let ((t1 (current-time-string)))
    (while (> (setq n (1- n))
              0))
    (list t1 (current-time-string))))
     ⇒ silly-loop

(disassemble 'silly-loop)
     ⊣ byte-code for silly-loop:
 doc: Return time before and after N iterations of a loop.
 args: (arg1)

0   constant current-time-string  ; Push current-time-string
                                  ;    onto top of stack.
1   call     0             ; Call current-time-string with no
                           ;    argument, push result onto stack as t1.
2:1 stack-ref 1            ; Get value of the argument n
                           ;    and push the value on the stack.
3   sub1                   ; Subtract 1 from top of stack.
4   dup                    ; Duplicate top of stack; i.e., copy the top
                           ;    of the stack and push copy onto stack.
5   stack-set 3            ; Pop the top of the stack,
                           ;    and set n to the value.

;; (In effect, the sequence dup stack-set copies the top of
;; the stack into the value of n without popping it.)
```

```
7   constant 0              ; Push 0 onto stack.
8   gtr                     ; Pop top two values off stack,
                            ;   test if n is greater than 0
                            ;   and push result onto stack.
9   goto-if-not-nil 1       ; Goto 1 if n > 0
                            ;   (this continues the while loop)
                            ;   else continue.
12  dup                     ; Push value of t1 onto stack.
13  constant current-time-string  ; Push current-time-string
                                  ;   onto the top of the stack.
14  call     0              ; Call current-time-string again.
15  list2                   ; Pop top two elements off stack, create a
                            ;   list of them, and push it onto stack.
16  return                  ; Return value of the top of stack.
```

18 Compilation of Lisp to Native Code

In addition to the byte-compilation, described in Chapter 17 [Byte Compilation], page 317, Emacs can also optionally compile Lisp function definitions into a true compiled code, known as *native code*. This feature uses the `libgccjit` library, which is part of the GCC distribution, and requires that Emacs be built with support for using that library. It also requires to have GCC and Binutils (the assembler and linker) available on your system for you to be able to native-compile Lisp code.

To determine whether the current Emacs process can produce and load natively-compiled Lisp code, call `native-comp-available-p` (see Section 18.1 [Native-Compilation Functions], page 329).

Unlike byte-compiled code, natively-compiled Lisp code is executed directly by the machine's hardware, and therefore runs at full speed that the host CPU can provide. The resulting speedup generally depends on what the Lisp code does, but is usually 2.5 to 5 times faster than the corresponding byte-compiled code.

Since native code is generally incompatible between different systems, the natively-compiled code is *not* transportable from one machine to another, it can only be used on the same machine where it was produced or on very similar ones (having the same CPU and run-time libraries). The transportability of natively-compiled code is the same as that of shared libraries (`.so` or `.dll` files).

Libraries of natively-compiled code include crucial dependencies on Emacs Lisp primitives (see Section 13.1 [What Is a Function], page 230) and their calling conventions, and thus Emacs usually won't load natively-compiled code produced by earlier or later Emacs versions; native compilation of the same Lisp code by a different Emacs version will usually produce a natively-compiled library under a unique file name that only that version of Emacs will be able to load. However, the use of unique file names allows to have in the same directory several versions of the same Lisp library natively-compiled by several different versions of Emacs.

A non-`nil` file-local variable binding of `no-byte-compile` (see Chapter 17 [Byte Compilation], page 317) also disables the native compilation of that file. In addition, a similar variable `no-native-compile` disables just the native compilation of the file. If both `no-byte-compile` and `no-native-compile` are specified, the former takes precedence.

Sometimes there could be a need to prevent the native compilation from writing its results, the `*.eln` files, into a subdirectory of `user-emacs-directory` (see Section 42.1.2 [Init File], page 1263). You can do that by either changing the value of `native-comp-eln-load-path` (see Section 18.2 [Native-Compilation Variables], page 331) or by temporarily pointing the `HOME` environment variable to a non-existing directory. Note that the latter technique might still produce a small number of `*.eln` files if Emacs needs to generate *trampolines*, which are used if Lisp primitives are advised or redefined in your Lisp code that is being natively compiled. See Section 18.2 [Native-Compilation Variables], page 331. Alternatively, you can specify that the `*.eln` files are written to a non-default directory using the `startup-redirect-eln-cache` function; see Section 18.1 [Native-Compilation Functions], page 329.

18.1 Native-Compilation Functions

Native-Compilation is implemented as a side effect of byte-compilation (see Chapter 17 [Byte Compilation], page 317). Thus, compiling Lisp code natively always produces its byte code as well, and therefore all the rules and caveats of preparing Lisp code for byte compilation (see Section 17.2 [Compilation Functions], page 317) are valid for native-compilation as well.

You can natively-compile either a single function or macro definition, or a whole file of Lisp code, with the `native-compile` function. Natively-compiling a file will produce the `.eln` file with native code.

Native compilation might produce warning or error messages; these are normally recorded in the buffer called `*Native-compile-Log*`. In interactive sessions, it uses the special LIMPLE mode (`native-comp-limple-mode`), which sets up `font-lock` as appropriate for this log, and is otherwise the same as Fundamental mode. Logging of messages resulting from native-compilation can be controlled by the `native-comp-verbose` variable (see Section 18.2 [Native-Compilation Variables], page 331).

When Emacs is run non-interactively, messages produced by native-compilation are reported by calling `message` (see Section 41.4.1 [Displaying Messages], page 1135), and are usually displayed on the standard error stream of the terminal from which Emacs was invoked.

`native-compile` *function-or-file* **&optional** *output* [Function]

This function compiles *function-or-file* into native code. The argument *function-or-file* can be a function symbol, a Lisp form, or a name (a string) of the file which contains the Emacs Lisp source code to compile. If the optional argument *output* is provided, it must be a string specifying the name of the file to write the compiled code into. Otherwise, if *function-or-file* is a function or a Lisp form, this function returns the compiled object, and if *function-or-file* is a file name, the function returns the full absolute name of the file it created for the compiled code. The output file is by default given the `.eln` extension.

This function runs the final phase of the native compilation, which invokes GCC via `libgccjit`, in a separate subprocess, which invokes the same Emacs executable as the process that called this function.

`batch-native-compile` **&optional** *for-tarball* [Function]

This function runs native-compilation on files specified on the Emacs command line in batch mode. It must be used only in a batch execution of Emacs, as it kills Emacs upon completion of the compilation. If one or more of the files fail to compile, the Emacs process will attempt to compile all the other files, and will terminate with a non-zero status code. The optional argument *for-tarball*, if non-`nil`, tells the function to place the resulting `.eln` files in the last directory mentioned in `native-comp-eln-load-path` (see Section 16.3 [Library Search], page 302); this is meant to be used as part of building an Emacs source tarball for the first time, when the natively-compiled files, which are absent from the source tarball, should be generated in the build tree instead of the user's cache directory.

Native compilation can be run entirely asynchronously, in a subprocess of the main Emacs process. This leaves the main Emacs process free to use while the compilation

runs in the background. This is the method used by Emacs to natively-compile any Lisp file or byte-compiled Lisp file that is loaded into Emacs, when no natively-compiled file for it is available. Note that because of this use of a subprocess, native compilation may produce warning and errors which byte-compilation does not, and Lisp code may thus need to be modified to work correctly. See `native-comp-async-report-warnings-errors` in see Section 18.2 [Native-Compilation Variables], page 331, for more details.

`native-compile-async` *files* **&optional** *recursively load selector* [Function]
: This function compiles the named *files* asynchronously. The argument *files* should be a single file name (a string) or a list of one or more file and/or directory names. If directories are present in the list, the optional argument *recursively* should be non-`nil` to cause the compilation to recurse into those directories. If *load* is non-`nil`, Emacs will load each file that it succeeded to compile. The optional argument *selector* allows control of which of *files* will be compiled; it can have one of the following values:

  `nil` or omitted
  : Select all the files and directories in *files*.

  a regular expression string
  : Select the files and directories whose names match the regexp.

  a function
  : A predicate function, which will be called with each file and directory in *files*, and should return non-`nil` if the file or the directory should be selected for compilation.

  On systems with multiple CPU execution units, when *files* names more than one file, this function will normally start several compilation subprocesses in parallel, under the control of `native-comp-async-jobs-number` (see Section 18.2 [Native-Compilation Variables], page 331).

`emacs-lisp-native-compile` [Command]
: This command compiles the file visited by the current buffer into native code, if the file was changed since the last time it was natively-compiled.

`emacs-lisp-native-compile-and-load` [Command]
: This command compiles the file visited by the current buffer into native code, like `emacs-lisp-native-compile`, but it also loads the native code when the compilation finishes.

The following function allows Lisp programs to test whether native-compilation is available at runtime.

`native-comp-available-p` [Function]
: This function returns non-`nil` if the running Emacs process has the native-compilation support compiled into it. On systems that load `libgccjit` dynamically, it also makes sure that library is available and can be loaded. Lisp programs that need to know up front whether native-compilation is available should use this predicate.

By default, asynchronous native compilation writes the `*.eln` files it produces to a subdirectory of the first writable directory specified by the `native-comp-eln-load-path` variable (see Section 18.2 [Native-Compilation Variables], page 331). You can change this by using the following function in your startup files:

`startup-redirect-eln-cache` *cache-directory* [Function]
: This function arranges for the asynchronous native compilation to write the produced `*.eln` files to *cache-directory*, which must be a single directory, a string. It also destructively modifies `native-comp-eln-load-path` such that its first element is *cache-directory*. If *cache-directory* is not an absolute file name, it is interpreted relative to `user-emacs-directory` (see Section 42.1.2 [Init File], page 1263).

18.2 Native-Compilation Variables

This section documents the variables that control native-compilation.

`native-comp-speed` [User Option]
: This variable specifies the optimization level for native compilation. Its value should be a number between −1 and 3. Values between 0 and 3 specify the optimization levels equivalent to the corresponding compiler `-O0`, `-O1`, etc. command-line options of the compiler. The value −1 means disable native-compilation: functions and files will be only byte-compiled; however, the `*.eln` files will still be produced, they will just contain the compiled code in bytecode form. (This can be achieved at function granularity by using the `(declare (speed -1))` form, see Section 13.15 [Declare Form], page 263.) The default value is 2.

`native-comp-debug` [User Option]
: This variable specifies the level of debugging information produced by native-compilation. Its value should be a number between zero and 3, with the following meaning:

  0 — No debugging output. This is the default.

  1 — Emit debugging symbols with the native code. This allows easier debugging of the native code with debuggers such as `gdb`.

  2 — Like 1, and in addition dump pseudo-C code.

  3 — Like 2, and in addition dump the GCC intermediate passes and `libgccjit` log file.

`native-comp-verbose` [User Option]
: This variable controls the verbosity of native-compilation by suppressing some or all of the log messages emitted by it. If its value is zero, the default, all of the log messages are suppressed. Setting it to a value between 1 and 3 will allow logging of the messages whose level is above the value. The values have the following interpretations:

  0 — No logging. This is the default.

  1 — Log the final LIMPLE representation of the code.

  2 — Log the LAP, the final LIMPLE, and some additional pass info.

  3 — Maximum verbosity: log everything.

`native-comp-async-jobs-number` [User Option]
: This variable determines the maximum number of native-compilation subprocesses that will be started simultaneously. It should be a non-negative number. The default value is zero, which means use half the number of the CPU execution units, or 1 if the CPU has only one execution unit.

`native-comp-async-report-warnings-errors` [User Option]
: If this variable's value is non-`nil`, warnings and errors from asynchronous native-compilation subprocesses are reported in the main Emacs session in a buffer named `*Warnings*`. The default value `t` means display the resulting buffer. To log warnings without popping up the `*Warnings*` buffer, set this variable to `silent`.

  A common cause for asynchronous native-compilation to produce warnings is compiling a file that is missing some `require` of a necessary feature. The feature may be loaded into the main emacs, but because native compilation always starts from a subprocess with a pristine environment, that may not be true for the subprocess.

`native-comp-async-query-on-exit` [User Option]
: If this variable's value is non-`nil`, Emacs will query upon exiting whether to exit and kill any asynchronous native-compilation subprocesses that are still running, thus preventing the corresponding `.eln` files from being written. If the value is `nil`, the default, Emacs will kill these subprocesses without querying.

The variable `native-comp-eln-load-path` holds the list of directories where Emacs looks for the `*.eln` files (see Section 16.3 [Library Search], page 302); in that role it is the equivalent of `load-path` used to look for `*.el` and `*.elc` files. The directories in this list are also used for writing the `*.eln` files produced by asynchronous native-compilation; specifically, Emacs will write these files into the first writable directory in the list. Thus, you can control where native-compilation stores the results by changing the value of this variable.

`native-comp-jit-compilation` [Variable]
: This variable, if non-`nil`, enables asynchronous (a.k.a. *just-in-time*, or JIT) native compilation of the `*.elc` files loaded by Emacs for which the corresponding `*.eln` files do not already exist. This JIT compilation uses separate Emacs sub-processes running in batch mode, according to the value of `native-comp-async-jobs-number`. When the JIT compilation of a Lisp file finishes successfully, the resulting `.eln` file is loaded and its code replaces the definition of functions provided by the `.elc` file.

Setting the value of `native-comp-jit-compilation` to `nil` disables JIT native compilation. However, even when JIT native compilation is disabled, Emacs might still need to start asynchronous native compilation subprocesses to produce *trampolines*. To control this, use a separate variable, described below.

`native-comp-enable-subr-trampolines` [Variable]
: This variable controls generation of trampolines. A trampoline is a small piece of native code required to allow calling Lisp primitives, which were advised or redefined, from Lisp code that was natively-compiled with `native-comp-speed` set to 2 or greater. Emacs stores the generated trampolines on separate `*.eln` files. By default, this variable's value is `t`, which enables the generation of trampoline files; setting it to `nil` disables the generation of trampolines. Note that if a trampoline needed for advising or redefining a primitive is not available and cannot be generated, calls to that primitive from natively-compiled Lisp will ignore redefinitions and advices, and will behave as if the primitive was called directly from C. Therefore, we don't recommend disabling the trampoline generation, unless you know that all the

trampolines needed by your Lisp programs are already compiled and accessible to Emacs.

The value of this variable can also be a string, in which case it specifies the name of a directory in which to store the generated trampoline `*.eln` files, overriding the directories in `native-comp-eln-load-path`. This is useful if you want the trampolines to be generated as needed, but don't want to store them under the user's `HOME` directory or in the other public directories where `*.eln` files are kept. However, unlike with directories in `native-comp-eln-load-path`, the trampolines will be stored in the directory given by the value of this variable, not in its version-specific subdirectory. If the name of this directory is not absolute, it is interpreted relative to `invocation-directory` (see Section 42.3 [System Environment], page 1270)

If this variable is non-`nil`, and Emacs needs to produce a trampoline, but it cannot find any writable directory to store the trampoline, it will store it inside `temporary-file-directory` (see Section 26.9.5 [Unique File Names], page 644).

Trampolines produced when no writable directory is found to store them, or when this variable is a string, will only be available for the duration of the current Emacs session, because Emacs doesn't look for trampolines in either of these places.

19 Debugging Lisp Programs

There are several ways to find and investigate problems in an Emacs Lisp program.

- If a problem occurs when you run the program, you can use the built-in Emacs Lisp debugger (see Section 19.1 [Debugger], page 334) to suspend the Lisp evaluator, and examine and/or alter its internal state.
- You can use Edebug, a source-level debugger for Emacs Lisp. See Section 19.2 [Edebug], page 345.
- You can trace the execution of functions involved in the problem using the tracing facilities provided by the `trace.el` package. This package provides the functions `trace-function-foreground` and `trace-function-background` for tracing function calls, and `trace-values` for adding values of select variables to the trace. For the details, see the documentation of these facilities in `trace.el`.
- If a syntactic problem is preventing Lisp from even reading the program, you can locate it using Lisp editing commands.
- You can look at the error and warning messages produced by the byte compiler when it compiles the program. See Section 17.6 [Compiler Errors], page 322.
- You can use the Testcover package to perform coverage testing on the program.
- You can use the ERT package to write regression tests for the program. See *ERT: Emacs Lisp Regression Testing*.
- You can profile the program to get hints about how to make it more efficient. See Section 19.5 [Profiling], page 370.

Other useful tools for debugging input and output problems are the dribble file (see Section 42.13 [Terminal Input], page 1290) and the `open-termscript` function (see Section 42.14 [Terminal Output], page 1292).

19.1 The Lisp Debugger

The ordinary *Lisp debugger* provides the ability to suspend evaluation of a form. While evaluation is suspended (a state that is commonly known as a *break*), you may examine the run time stack, examine the values of local or global variables, or change those values. Since a break is a recursive edit, all the usual editing facilities of Emacs are available; you can even run programs that will enter the debugger recursively. See Section 22.13 [Recursive Editing], page 476.

19.1.1 Entering the Debugger on an Error

The most important time to enter the debugger is when a Lisp error happens. This allows you to investigate the immediate causes of the error.

However, entry to the debugger is not a normal consequence of an error. Many commands signal Lisp errors when invoked inappropriately, and during ordinary editing it would be very inconvenient to enter the debugger each time this happens. So if you want errors to enter the debugger, set the variable `debug-on-error` to non-`nil`. (The command `toggle-debug-on-error` provides an easy way to do this.)

Note that, for technical reasons, you cannot use the facilities defined in this subsection to debug errors in Lisp that the redisplay code has invoked. See Section 19.1.2 [Debugging Redisplay], page 336, for help with these.

`debug-on-error` [User Option]

This variable determines whether the debugger is called when an error is signaled and not handled. If `debug-on-error` is `t`, all kinds of errors call the debugger, except those listed in `debug-ignored-errors` (see below). If it is `nil`, none call the debugger.

The value can also be a list of error conditions (see Section 11.7.3.1 [Signaling Errors], page 180). Then the debugger is called only for error conditions in this list (except those also listed in `debug-ignored-errors`). For example, if you set `debug-on-error` to the list `(void-variable)`, the debugger is only called for errors about a variable that has no value.

Note that `eval-expression-debug-on-error` overrides this variable in some cases; see below.

When this variable is non-`nil`, Emacs does not create an error handler around process filter functions and sentinels. Therefore, errors in these functions also invoke the debugger. See Chapter 40 [Processes], page 1081.

`debug-ignored-errors` [User Option]

This variable specifies errors which should not enter the debugger, regardless of the value of `debug-on-error`. Its value is a list of error condition symbols and/or regular expressions. If the error has any of those condition symbols, or if the error message matches any of the regular expressions, then that error does not enter the debugger.

The normal value of this variable includes `user-error`, as well as several errors that happen often during editing but rarely result from bugs in Lisp programs. However, "rarely" is not "never"; if your program fails with an error that matches this list, you may try changing this list to debug the error. The easiest way is usually to set `debug-ignored-errors` to `nil`.

`eval-expression-debug-on-error` [User Option]

If this variable has a non-`nil` value (the default), running the command `eval-expression` causes `debug-on-error` to be temporarily bound to `t`. See Section "Evaluating Emacs Lisp Expressions" in *The GNU Emacs Manual*.

If `eval-expression-debug-on-error` is `nil`, then the value of `debug-on-error` is not changed during `eval-expression`.

`debug-on-signal` [User Option]

Normally, errors caught by `condition-case` never invoke the debugger. The `condition-case` gets a chance to handle the error before the debugger gets a chance.

If you change `debug-on-signal` to a non-`nil` value, the debugger gets the first chance at every error, regardless of the presence of `condition-case`. (To invoke the debugger, the error must still fulfill the criteria specified by `debug-on-error` and `debug-ignored-errors`.)

For example, setting this variable is useful to get a backtrace from code evaluated by emacsclient's `--eval` option. If Lisp code evaluated by emacsclient signals an error while this variable is non-`nil`, the backtrace will popup in the running Emacs.

Warning: Setting this variable to non-`nil` may have annoying effects. Various parts of Emacs catch errors in the normal course of affairs, and you may not even realize that errors happen there. If you need to debug code wrapped in `condition-case`, consider using `condition-case-unless-debug` (see Section 11.7.3.3 [Handling Errors], page 182).

`debug-on-event` [User Option]

If you set `debug-on-event` to a special event (see Section 22.9 [Special Events], page 471), Emacs will try to enter the debugger as soon as it receives this event, bypassing `special-event-map`. At present, the only supported values correspond to the signals `SIGUSR1` and `SIGUSR2` (this is the default). This can be helpful when `inhibit-quit` is set and Emacs is not otherwise responding.

`debug-on-message` [Variable]

If you set `debug-on-message` to a regular expression, Emacs will enter the debugger if it displays a matching message in the echo area. For example, this can be useful when trying to find the cause of a particular message.

`debug-allow-recursive-debug` [Variable]

You can evaluate forms in the current stack frame in the '`*Backtrace*`' buffer with the `e` command, and while edebugging you can use the `e` and `C-x C-e` commands to do something similar. By default, the debugger is inhibited by these commands (because (re-)entering the debugger at this point will usually take you out of the debugging context you're in). Set `debug-allow-recursive-debug` to a non-`nil` value to allow these commands to enter the debugger recursively.

To debug an error that happens during loading of the init file, use the option '`--debug-init`'. This binds `debug-on-error` to `t` while loading the init file, and bypasses the `condition-case` which normally catches errors in the init file.

19.1.2 Debugging Redisplay Errors

When an error occurs in Lisp code which redisplay has invoked, Emacs's usual debugging mechanisms are unusable, for technical reasons. This subsection describes how to get a backtrace from such an error, which should be helpful in debugging it.

These directions apply to Lisp forms used, for example, in `:eval` mode line constructs (see Section 24.4.2 [Mode Line Data], page 552), and in all hooks invoked from redisplay, such as:

- `fontification-functions` (see Section 41.12.7 [Auto Faces], page 1182).
- `window-scroll-functions` (see Section 29.28 [Window Hooks], page 782).

Note that if you have had an error in a hook function called from redisplay, the error handling might have removed this function from the hook. You will thus need to reinitialize that hook somehow, perhaps with `add-hook`, to be able to replay the bug.

To generate a backtrace in these circumstances, set the variable `backtrace-on-redisplay-error` to non-`nil`. When the error occurs, Emacs will dump the backtrace to the buffer `*Redisplay-trace*`, but won't automatically display it in a window. This is to avoid needlessly corrupting the redisplay you are debugging. You will thus need to display the buffer yourself, with a command such as `switch-to-buffer-other-frame` `C-x 5 b`.

`backtrace-on-redisplay-error` [Variable]

Set this variable to non-`nil` to enable the generation of a backtrace when an error occurs in any Lisp called from redisplay.

19.1.3 Debugging Infinite Loops

When a program loops infinitely and fails to return, your first problem is to stop the loop. On most operating systems, you can do this with `C-g`, which causes a *quit*. See Section 22.11 [Quitting], page 472.

Ordinary quitting gives no information about why the program was looping. To get more information, you can set the variable `debug-on-quit` to non-`nil`. Once you have the debugger running in the middle of the infinite loop, you can proceed from the debugger using the stepping commands. If you step through the entire loop, you may get enough information to solve the problem.

Quitting with `C-g` is not considered an error, and `debug-on-error` has no effect on the handling of `C-g`. Likewise, `debug-on-quit` has no effect on errors.

`debug-on-quit` [User Option]

This variable determines whether the debugger is called when `quit` is signaled and not handled. If `debug-on-quit` is non-`nil`, then the debugger is called whenever you quit (that is, type `C-g`). If `debug-on-quit` is `nil` (the default), then the debugger is not called when you quit.

19.1.4 Entering the Debugger on a Function Call

To investigate a problem that happens in the middle of a program, one useful technique is to enter the debugger whenever a certain function is called. You can do this to the function in which the problem occurs, and then step through the function, or you can do this to a function called shortly before the problem, step quickly over the call to that function, and then step through its caller.

`debug-on-entry` *function-name* [Command]

This function requests *function-name* to invoke the debugger each time it is called.

Any function or macro defined as Lisp code may be set to break on entry, regardless of whether it is interpreted code or compiled code. If the function is a command, it will enter the debugger when called from Lisp and when called interactively (after the reading of the arguments). You can also set debug-on-entry for primitive functions (i.e., those written in C) this way, but it only takes effect when the primitive is called from Lisp code. Debug-on-entry is not allowed for special forms.

When `debug-on-entry` is called interactively, it prompts for *function-name* in the minibuffer. If the function is already set up to invoke the debugger on entry, `debug-on-entry` does nothing. `debug-on-entry` always returns *function-name*.

Here's an example to illustrate use of this function:

```
(defun fact (n)
  (if (zerop n) 1
      (* n (fact (1- n)))))
     ⇒ fact
(debug-on-entry 'fact)
     ⇒ fact
```

```
(fact 3)

------ Buffer: *Backtrace* ------
Debugger entered--entering a function:
* fact(3)
  eval((fact 3))
  eval-last-sexp-1(nil)
  eval-last-sexp(nil)
  call-interactively(eval-last-sexp)
------ Buffer: *Backtrace* ------
```

`cancel-debug-on-entry` **&optional** *function-name* [Command]
: This function undoes the effect of `debug-on-entry` on *function-name*. When called interactively, it prompts for *function-name* in the minibuffer. If *function-name* is omitted or `nil`, it cancels break-on-entry for all functions. Calling `cancel-debug-on-entry` does nothing to a function which is not currently set up to break on entry.

19.1.5 Entering the debugger when a variable is modified

Sometimes a problem with a function is due to a wrong setting of a variable. Setting up the debugger to trigger whenever the variable is changed is a quick way to find the origin of the setting.

`debug-on-variable-change` *variable* [Command]
: This function arranges for the debugger to be called whenever *variable* is modified.

  It is implemented using the watchpoint mechanism, so it inherits the same characteristics and limitations: all aliases of *variable* will be watched together, only dynamic variables can be watched, and changes to the objects referenced by variables are not detected. For details, see Section 12.9 [Watching Variables], page 200.

`cancel-debug-on-variable-change` **&optional** *variable* [Command]
: This function undoes the effect of `debug-on-variable-change` on *variable*. When called interactively, it prompts for *variable* in the minibuffer. If *variable* is omitted or `nil`, it cancels break-on-change for all variables. Calling `cancel-debug-on-variable-change` does nothing to a variable which is not currently set up to break on change.

19.1.6 Explicit Entry to the Debugger

You can cause the debugger to be called at a certain point in your program by writing the expression `(debug)` at that point. To do this, visit the source file, insert the text '`(debug)`' at the proper place, and type *C-M-x* (`eval-defun`, a Lisp mode key binding). **Warning:** if you do this for temporary debugging purposes, be sure to undo this insertion before you save the file!

The place where you insert '`(debug)`' must be a place where an additional form can be evaluated and its value ignored. (If the value of `(debug)` isn't ignored, it will alter the execution of the program!) The most common suitable places are inside a `progn` or an implicit `progn` (see Section 11.1 [Sequencing], page 158).

If you don't know exactly where in the source code you want to put the debug statement, but you want to display a backtrace when a certain message is displayed, you can set `debug-on-message` to a regular expression matching the desired message.

19.1.7 Using the Debugger

When the debugger is entered, it displays the previously selected buffer in one window and a buffer named `*Backtrace*` in another window. The backtrace buffer contains one line for each level of Lisp function execution currently going on. At the beginning of this buffer is a message describing the reason that the debugger was invoked (such as the error message and associated data, if it was invoked due to an error).

The backtrace buffer is read-only and uses a special major mode, Debugger mode, in which letters are defined as debugger commands. The usual Emacs editing commands are available; thus, you can switch windows to examine the buffer that was being edited at the time of the error, switch buffers, visit files, or do any other sort of editing. However, the debugger is a recursive editing level (see Section 22.13 [Recursive Editing], page 476) and it is wise to go back to the backtrace buffer and exit the debugger (with the `q` command) when you are finished with it. Exiting the debugger gets out of the recursive edit and buries the backtrace buffer. (You can customize what the `q` command does with the backtrace buffer by setting the variable `debugger-bury-or-kill`. For example, set it to `kill` if you prefer to kill the buffer rather than bury it. Consult the variable's documentation for more possibilities.)

When the debugger has been entered, the `debug-on-error` variable is temporarily set according to `eval-expression-debug-on-error`. If the latter variable is non-`nil`, `debug-on-error` will temporarily be set to `t`. However, further errors that occur while debugging won't (by default) trigger another debugger, because `inhibit-debugger` will also be bound to non-`nil`.

The debugger itself must be run byte-compiled, since it makes assumptions about the state of the Lisp interpreter. These assumptions are false if the debugger is running interpreted.

19.1.8 Backtraces

Debugger mode is derived from Backtrace mode, which is also used to show backtraces by Edebug and ERT. (see Section 19.2 [Edebug], page 345, and *ERT: Emacs Lisp Regression Testing*.)

The backtrace buffer shows you the functions that are executing and their argument values. When a backtrace buffer is created, it shows each stack frame on one, possibly very long, line. (A stack frame is the place where the Lisp interpreter records information about a particular invocation of a function.) The most recently called function will be at the top.

In a backtrace you can specify a stack frame by moving point to a line describing that frame. The frame whose line point is on is considered the *current frame*.

If a function name is underlined, that means Emacs knows where its source code is located. You can click with the mouse on that name, or move to it and type `RET`, to visit the source code. You can also type `RET` while point is on any name of a function or variable which is not underlined, to see help information for that symbol in a help buffer, if any exists. The `xref-find-definitions` command, bound to `M-.`, can also be used on any identifier in a backtrace (see Section "Looking Up Identifiers" in *The GNU Emacs Manual*).

In backtraces, the tails of long lists and the ends of long strings, vectors or structures, as well as objects which are deeply nested, will be printed as underlined "...". You can click with the mouse on a "...", or type `RET` while point is on it, to show the part of the object that was hidden. To control how much abbreviation is done, customize `backtrace-line-length`.

Here is a list of commands for navigating and viewing backtraces:

`v` Toggle the display of local variables of the current stack frame.

`p` Move to the beginning of the frame, or to the beginning of the previous frame.

`n` Move to the beginning of the next frame.

`+` Add line breaks and indentation to the top-level Lisp form at point to make it more readable.

`-` Collapse the top-level Lisp form at point back to a single line.

`#` Toggle `print-circle` for the frame at point.

`:` Toggle `print-gensym` for the frame at point.

`.` Expand all the forms abbreviated with "..." in the frame at point.

19.1.9 Debugger Commands

The debugger buffer (in Debugger mode) provides special commands in addition to the usual Emacs commands and to the Backtrace mode commands described in the previous section. The most important use of debugger commands is for stepping through code, so that you can see how control flows. The debugger can step through the control structures of an interpreted function, but cannot do so in a byte-compiled function. If you would like to step through a byte-compiled function, replace it with an interpreted definition of the same function. (To do this, visit the source for the function and type `C-M-x` on its definition.) You cannot use the Lisp debugger to step through a primitive function.

Some of the debugger commands operate on the current frame. If a frame starts with a star, that means that exiting that frame will call the debugger again. This is useful for examining the return value of a function.

Here is a list of Debugger mode commands:

`c` Exit the debugger and continue execution. This resumes execution of the program as if the debugger had never been entered (aside from any side-effects that you caused by changing variable values or data structures while inside the debugger).

`d` Continue execution, but enter the debugger the next time any Lisp function is called. This allows you to step through the subexpressions of an expression, seeing what values the subexpressions compute, and what else they do.

The stack frame made for the function call which enters the debugger in this way will be flagged automatically so that the debugger will be called again when the frame is exited. You can use the `u` command to cancel this flag.

`b` Flag the current frame so that the debugger will be entered when the frame is exited. Frames flagged in this way are marked with stars in the backtrace buffer.

`u` Don't enter the debugger when the current frame is exited. This cancels a `b` command on that frame. The visible effect is to remove the star from the line in the backtrace buffer.

`j` Flag the current frame like `b`. Then continue execution like `c`, but temporarily disable break-on-entry for all functions that are set up to do so by `debug-on-entry`.

`e` Read a Lisp expression in the minibuffer, evaluate it (with the relevant lexical environment, if applicable), and print the value in the echo area. The debugger alters certain important variables, and the current buffer, as part of its operation; `e` temporarily restores their values from outside the debugger, so you can examine and change them. This makes the debugger more transparent. By contrast, `M-:` does nothing special in the debugger; it shows you the variable values within the debugger. By default, this command suppresses the debugger during evaluation, so that an error in the evaluated expression won't add a new error on top of the existing one. Set the `debug-allow-recursive-debug` user option to a non-`nil` value to override this.

`R` Like `e`, but also save the result of evaluation in the buffer `*Debugger-record*`.

`q` Terminate the program being debugged; return to top-level Emacs command execution.

If the debugger was entered due to a `C-g` but you really want to quit, and not debug, use the `q` command.

`r` Return a value from the debugger. The value is computed by reading an expression with the minibuffer and evaluating it.

The `r` command is useful when the debugger was invoked due to exit from a Lisp call frame (as requested with `b` or by entering the frame with `d`); then the value specified in the `r` command is used as the value of that frame. It is also useful if you call `debug` and use its return value. Otherwise, `r` has the same effect as `c`, and the specified return value does not matter.

You can't use `r` when the debugger was entered due to an error.

`l` Display a list of functions that will invoke the debugger when called. This is a list of functions that are set to break on entry by means of `debug-on-entry`.

19.1.10 Invoking the Debugger

Here we describe in full detail the function `debug` that is used to invoke the debugger.

`debug` **&rest** *debugger-args* [Command]

This function enters the debugger. In interactive sessions, it switches to a buffer named `*Backtrace*` (or `*Backtrace*<2>` if it is the second recursive entry to the debugger, etc.), and fills it with information about the stack of Lisp function calls. It then enters a recursive edit, showing the backtrace buffer in Debugger mode. In batch mode (more generally, when `noninteractive` is non-`nil`, see Section 42.17 [Batch Mode], page 1294), this function shows the Lisp backtrace on the standard error stream, and then kills Emacs, causing it to exit with a non-zero exit code

(see Section 42.2.1 [Killing Emacs], page 1267). Binding `backtrace-on-error-noninteractive` to `nil` suppresses the backtrace in batch mode, see below.

The Debugger mode `c`, `d`, `j`, and `r` commands exit the recursive edit; then `debug` switches back to the previous buffer and returns to whatever called `debug`. This is the only way the function `debug` can return to its caller.

The use of the *debugger-args* is that `debug` displays the rest of its arguments at the top of the `*Backtrace*` buffer, so that the user can see them. Except as described below, this is the *only* way these arguments are used.

However, certain values for first argument to `debug` have a special significance. (Normally, these values are used only by the internals of Emacs, and not by programmers calling `debug`.) Here is a table of these special values:

`lambda` A first argument of `lambda` means `debug` was called because of entry to a function when `debug-on-next-call` was non-`nil`. The debugger displays '`Debugger entered--entering a function:`' as a line of text at the top of the buffer.

`debug` `debug` as first argument means `debug` was called because of entry to a function that was set to debug on entry. The debugger displays the string '`Debugger entered--entering a function:`', just as in the `lambda` case. It also marks the stack frame for that function so that it will invoke the debugger when exited.

`t` When the first argument is `t`, this indicates a call to `debug` due to evaluation of a function call form when `debug-on-next-call` is non-`nil`. The debugger displays '`Debugger entered--beginning evaluation of function call form:`' as the top line in the buffer.

`exit` When the first argument is `exit`, it indicates the exit of a stack frame previously marked to invoke the debugger on exit. The second argument given to `debug` in this case is the value being returned from the frame. The debugger displays '`Debugger entered--returning value:`' in the top line of the buffer, followed by the value being returned.

`error` When the first argument is `error`, the debugger indicates that it is being entered because an error or `quit` was signaled and not handled, by displaying '`Debugger entered--Lisp error:`' followed by the error signaled and any arguments to `signal`. For example,

```
(let ((debug-on-error t))
  (/ 1 0))

------ Buffer: *Backtrace* ------
Debugger entered--Lisp error: (arith-error)
  /(1 0)
...
------ Buffer: *Backtrace* ------
```

If an error was signaled, presumably the variable `debug-on-error` is non-`nil`. If `quit` was signaled, then presumably the variable `debug-on-quit` is non-`nil`.

`nil` Use `nil` as the first of the *debugger-args* when you want to enter the debugger explicitly. The rest of the *debugger-args* are printed on the top line of the buffer. You can use this feature to display messages—for example, to remind yourself of the conditions under which `debug` is called.

`backtrace-on-error-noninteractive` [Variable]
If this variable is non-`nil`, the default, entering the debugger in batch mode shows the backtrace of Lisp functions calls. Binding the variable to the `nil` value suppresses the backtrace and shows only the error message.

19.1.11 Internals of the Debugger

This section describes functions and variables used internally by the debugger.

`debugger` [Variable]
The value of this variable is the function to call to invoke the debugger. Its value must be a function of any number of arguments, or, more typically, the name of a function. This function should invoke some kind of debugger. The default value of the variable is `debug`.

The first argument that Lisp hands to the function indicates why it was called. The convention for arguments is detailed in the description of `debug` (see Section 19.1.10 [Invoking the Debugger], page 341).

`backtrace` [Function]
This function prints a trace of Lisp function calls currently active. The trace is identical to the one that `debug` would show in the `*Backtrace*` buffer. The return value is always `nil`.

In the following example, a Lisp expression calls `backtrace` explicitly. This prints the backtrace to the stream `standard-output`, which, in this case, is the buffer ‘`backtrace-output`’.

Each line of the backtrace represents one function call. The line shows the function followed by a list of the values of the function’s arguments if they are all known; if they are still being computed, the line consists of a list containing the function and its unevaluated arguments. Long lists or deeply nested structures may be elided.

```
(with-output-to-temp-buffer "backtrace-output"
  (let ((var 1))
    (save-excursion
      (setq var (eval '(progn
                        (1+ var)
                        (list 'testing (backtrace))))))))

     ⇒ (testing nil)

---------- Buffer: backtrace-output ------------
  backtrace()
  (list 'testing (backtrace))
  (progn ...)
  eval((progn (1+ var) (list 'testing (backtrace))))
  (setq ...)
  (save-excursion ...)
  (let ...)
```

```
  (with-output-to-temp-buffer ...)
  eval((with-output-to-temp-buffer ...))
  eval-last-sexp-1(nil)
  eval-last-sexp(nil)
  call-interactively(eval-last-sexp)
---------- Buffer: backtrace-output ------------
```

`debugger-stack-frame-as-list` [User Option]

If this variable is non-`nil`, every stack frame of the backtrace is displayed as a list. This aims at improving the backtrace readability at the cost of special forms no longer being visually different from regular function calls.

With `debugger-stack-frame-as-list` non-`nil`, the above example would look as follows:

```
---------- Buffer: backtrace-output ------------
  (backtrace)
  (list 'testing (backtrace))
  (progn ...)
  (eval (progn (1+ var) (list 'testing (backtrace))))
  (setq ...)
  (save-excursion ...)
  (let ...)
  (with-output-to-temp-buffer ...)
  (eval (with-output-to-temp-buffer ...))
  (eval-last-sexp-1 nil)
  (eval-last-sexp nil)
  (call-interactively eval-last-sexp)
---------- Buffer: backtrace-output ------------
```

`debug-on-next-call` [Variable]

If this variable is non-`nil`, it says to call the debugger before the next `eval`, `apply` or `funcall`. Entering the debugger sets `debug-on-next-call` to `nil`.

The `d` command in the debugger works by setting this variable.

`backtrace-debug` *level flag* [Function]

This function sets the debug-on-exit flag of the stack frame *level* levels down the stack, giving it the value *flag*. If *flag* is non-`nil`, this will cause the debugger to be entered when that frame later exits. Even a nonlocal exit through that frame will enter the debugger.

This function is used only by the debugger.

`command-debug-status` [Variable]

This variable records the debugging status of the current interactive command. Each time a command is called interactively, this variable is bound to `nil`. The debugger can set this variable to leave information for future debugger invocations during the same command invocation.

The advantage of using this variable rather than an ordinary global variable is that the data will never carry over to a subsequent command invocation.

This variable is obsolete and will be removed in future versions.

`backtrace-frame` *frame-number* **&optional** *base* [Function]

The function `backtrace-frame` is intended for use in Lisp debuggers. It returns information about what computation is happening in the stack frame *frame-number* levels down.

If that frame has not evaluated the arguments yet, or is a special form, the value is `(nil` *function arg-forms*`...)`.

If that frame has evaluated its arguments and called its function already, the return value is `(t` *function arg-values*`...)`.

In the return value, *function* is whatever was supplied as the CAR of the evaluated list, or a `lambda` expression in the case of a macro call. If the function has a `&rest` argument, that is represented as the tail of the list *arg-values*.

If *base* is specified, *frame-number* counts relative to the topmost frame whose *function* is *base*.

If *frame-number* is out of range, `backtrace-frame` returns `nil`.

`mapbacktrace` *function* **&optional** *base* [Function]

The function `mapbacktrace` calls *function* once for each frame in the backtrace, starting at the first frame whose function is *base* (or from the top if *base* is omitted or `nil`).

function is called with four arguments: *evald*, *func*, *args*, and *flags*.

If a frame has not evaluated its arguments yet or is a special form, *evald* is `nil` and *args* is a list of forms.

If a frame has evaluated its arguments and called its function already, *evald* is `t` and *args* is a list of values. *flags* is a plist of properties of the current frame: currently, the only supported property is `:debug-on-exit`, which is `t` if the stack frame's `debug-on-exit` flag is set.

19.2 Edebug

Edebug is a source-level debugger for Emacs Lisp programs, with which you can:

- Step through evaluation, stopping before and after each expression.
- Set conditional or unconditional breakpoints.
- Stop when a specified condition is true (the global break event).
- Trace slow or fast, stopping briefly at each stop point, or at each breakpoint.
- Display expression results and evaluate expressions as if outside of Edebug.
- Automatically re-evaluate a list of expressions and display their results each time Edebug updates the display.
- Output trace information on function calls and returns.
- Stop when an error occurs.
- Display a backtrace, omitting Edebug's own frames.
- Specify argument evaluation for macros and defining forms.
- Obtain rudimentary coverage testing and frequency counts.

The first three sections below should tell you enough about Edebug to start using it.

19.2.1 Using Edebug

To debug a Lisp program with Edebug, you must first *instrument* the Lisp code that you want to debug. A simple way to do this is to first move point into the definition of a function or macro and then do `C-u C-M-x` (`eval-defun` with a prefix argument). See Section 19.2.2 [Instrumenting], page 347, for alternative ways to instrument code.

Once a function is instrumented, any call to the function activates Edebug. Depending on which Edebug execution mode you have selected, activating Edebug may stop execution and let you step through the function, or it may update the display and continue execution while checking for debugging commands. The default execution mode is step, which stops execution. See Section 19.2.3 [Edebug Execution Modes], page 348.

Within Edebug, you normally view an Emacs buffer showing the source of the Lisp code you are debugging. This is referred to as the *source code buffer*, and it is temporarily read-only.

An arrow in the left fringe indicates the line where the function is executing. Point initially shows where within the line the function is executing, but this ceases to be true if you move point yourself.

If you instrument the definition of `fac` (shown below) and then execute `(fac 3)`, here is what you would normally see. Point is at the open-parenthesis before `if`.

```
(defun fac (n)
=>*(if (< 0 n)
       (* n (fac (1- n)))
    1))
```

The places within a function where Edebug can stop execution are called *stop points*. These occur both before and after each subexpression that is a list, and also after each variable reference. Here we use periods to show the stop points in the function `fac`:

```
(defun fac (n)
  .(if .(< 0 n.).
       .(* n. .(fac .(1- n.).).).
    1).)
```

The special commands of Edebug are available in the source code buffer in addition to the commands of Emacs Lisp mode. For example, you can type the Edebug command `SPC` to execute until the next stop point. If you type `SPC` once after entry to `fac`, here is the display you will see:

```
(defun fac (n)
=>(if *(< 0 n)
       (* n (fac (1- n)))
    1))
```

When Edebug stops execution after an expression, it displays the expression's value in the echo area.

Other frequently used commands are `b` to set a breakpoint at a stop point, `g` to execute until a breakpoint is reached, and `q` to exit Edebug and return to the top-level command loop. Type `?` to display a list of all Edebug commands.

19.2.2 Instrumenting for Edebug

In order to use Edebug to debug Lisp code, you must first *instrument* the code. Instrumenting code inserts additional code into it, to invoke Edebug at the proper places.

When you invoke command *C-M-x* (`eval-defun`) with a prefix argument on a function definition, it instruments the definition before evaluating it. (This does not modify the source code itself.) If the variable `edebug-all-defs` is non-`nil`, that inverts the meaning of the prefix argument: in this case, *C-M-x* instruments the definition *unless* it has a prefix argument. The default value of `edebug-all-defs` is `nil`. The command *M-x edebug-all-defs* toggles the value of the variable `edebug-all-defs`.

If `edebug-all-defs` is non-`nil`, then the commands `eval-region`, and `eval-buffer` also instrument any definitions they evaluate. Similarly, `edebug-all-forms` controls whether `eval-region` should instrument *any* form, even non-defining forms. This doesn't apply to loading or evaluations in the minibuffer. The command *M-x edebug-all-forms* toggles this option.

Another command, *M-x edebug-eval-top-level-form*, is available to instrument any top-level form regardless of the values of `edebug-all-defs` and `edebug-all-forms`. `edebug-defun` is an alias for `edebug-eval-top-level-form`.

While Edebug is active, the command *I* (`edebug-instrument-callee`) instruments the definition of the function or macro called by the list form after point, if it is not already instrumented. This is possible only if Edebug knows where to find the source for that function; for this reason, after loading Edebug, `eval-region` records the position of every definition it evaluates, even if not instrumenting it. See also the *i* command (see Section 19.2.4 [Jumping], page 349), which steps into the call after instrumenting the function.

Edebug knows how to instrument all the standard special forms, `interactive` forms with an expression argument, anonymous lambda expressions, and other defining forms. However, Edebug cannot determine on its own what a user-defined macro will do with the arguments of a macro call, so you must provide that information using Edebug specifications; for details, see Section 19.2.15 [Edebug and Macros], page 359.

When Edebug is about to instrument code for the first time in a session, it runs the hook `edebug-setup-hook`, then sets it to `nil`. You can use this to load Edebug specifications associated with a package you are using, but only when you use Edebug.

If Edebug detects a syntax error while instrumenting, it leaves point at the erroneous code and signals an `invalid-read-syntax` error. Example:

```
error  Invalid read syntax: "Expected lambda expression"
```

One potential reason for such a failure to instrument is that some macro definitions are not yet known to Emacs. To work around this, load the file which defines the function you are about to instrument.

To remove instrumentation from a definition, simply re-evaluate its definition in a way that does not instrument. There are two ways of evaluating forms that never instrument them: from a file with `load`, and from the minibuffer with `eval-expression` (*M-:*).

A different way to remove the instrumentation from a definition is to use the `edebug-remove-instrumentation` command. It also allows removing the instrumentation from everything that has been instrumented.

See Section 19.2.9 [Edebug Eval], page 353, for other evaluation functions available inside of Edebug.

19.2.3 Edebug Execution Modes

Edebug supports several execution modes for running the program you are debugging. We call these alternatives *Edebug execution modes*; do not confuse them with major or minor modes. The current Edebug execution mode determines how far Edebug continues execution before stopping—whether it stops at each stop point, or continues to the next breakpoint, for example—and how much Edebug displays the progress of the evaluation before it stops.

Normally, you specify the Edebug execution mode by typing a command to continue the program in a certain mode. Here is a table of these commands; all except for `S` resume execution of the program, at least for a certain distance.

`S` Stop: don't execute any more of the program, but wait for more Edebug commands (`edebug-stop`).

`SPC` Step: stop at the next stop point encountered (`edebug-step-mode`).

`n` Next: stop at the next stop point encountered after an expression (`edebug-next-mode`). Also see `edebug-forward-sexp` in Section 19.2.4 [Jumping], page 349.

`t` Trace: pause (normally one second) at each Edebug stop point (`edebug-trace-mode`).

`T` Rapid trace: update the display at each stop point, but don't actually pause (`edebug-Trace-fast-mode`).

`g` Go: run until the next breakpoint (`edebug-go-mode`). See Section 19.2.6.1 [Breakpoints], page 351.

`c` Continue: pause one second at each breakpoint, and then continue (`edebug-continue-mode`).

`C` Rapid continue: move point to each breakpoint, but don't pause (`edebug-Continue-fast-mode`).

`G` Go non-stop: ignore breakpoints (`edebug-Go-nonstop-mode`). You can still stop the program by typing `S`, or any editing command.

In general, the execution modes earlier in the above list run the program more slowly or stop sooner than the modes later in the list.

When you enter a new Edebug level, Edebug will normally stop at the first instrumented function it encounters. If you prefer to stop only at a break point, or not at all (for example, when gathering coverage data), change the value of `edebug-initial-mode` from its default `step` to `go`, or `Go-nonstop`, or one of its other values (see Section 19.2.16 [Edebug Options], page 365). You can do this readily with `C-x C-a C-m` (`edebug-set-initial-mode`):

`edebug-set-initial-mode` [Command]

This command, bound to `C-x C-a C-m`, sets `edebug-initial-mode`. It prompts you for a key to indicate the mode. You should enter one of the eight keys listed above, which sets the corresponding mode.

Note that you may reenter the same Edebug level several times if, for example, an instrumented function is called several times from one command.

While executing or tracing, you can interrupt the execution by typing any Edebug command. Edebug stops the program at the next stop point and then executes the command you typed. For example, typing `t` during execution switches to trace mode at the next stop point. You can use `S` to stop execution without doing anything else.

If your function happens to read input, a character you type intending to interrupt execution may be read by the function instead. You can avoid such unintended results by paying attention to when your program wants input.

Keyboard macros containing the commands in this section do not completely work: exiting from Edebug, to resume the program, loses track of the keyboard macro. This is not easy to fix. Also, defining or executing a keyboard macro outside of Edebug does not affect commands inside Edebug. This is usually an advantage. See also the `edebug-continue-kbd-macro` option in Section 19.2.16 [Edebug Options], page 365.

`edebug-sit-for-seconds` [User Option]

This option specifies how many seconds to wait between execution steps in trace mode or continue mode. The default is 1 second.

19.2.4 Jumping

The commands described in this section execute until they reach a specified location. All except `i` make a temporary breakpoint to establish the place to stop, then switch to go mode. Any other breakpoint reached before the intended stop point will also stop execution. See Section 19.2.6.1 [Breakpoints], page 351, for the details on breakpoints.

These commands may fail to work as expected in case of nonlocal exit, as that can bypass the temporary breakpoint where you expected the program to stop.

`h` Proceed to the stop point near where point is (`edebug-goto-here`).

`f` Run the program for one expression (`edebug-forward-sexp`).

`o` Run the program until the end of the containing sexp (`edebug-step-out`).

`i` Step into the function or macro called by the form after point (`edebug-step-in`).

The `h` command proceeds to the stop point at or after the current location of point, using a temporary breakpoint.

The `f` command runs the program forward over one expression. More precisely, it sets a temporary breakpoint at the position that `forward-sexp` would reach, then executes in go mode so that the program will stop at breakpoints.

With a prefix argument *n*, the temporary breakpoint is placed *n* sexps beyond point. If the containing list ends before *n* more elements, then the place to stop is after the containing expression.

You must check that the position `forward-sexp` finds is a place that the program will really get to. In `cond`, for example, this may not be true.

For flexibility, the `f` command does `forward-sexp` starting at point, rather than at the stop point. If you want to execute one expression *from the current stop point*, first type `w` (`edebug-where`) to move point there, and then type `f`.

The `o` command continues out of an expression. It places a temporary breakpoint at the end of the sexp containing point. If the containing sexp is a function definition itself, `o` continues until just before the last sexp in the definition. If that is where you are now, it returns from the function and then stops. In other words, this command does not exit the currently executing function unless you are positioned after the last sexp.

Normally, the `h`, `f`, and `o` commands display "Break" and pause for `edebug-sit-for-seconds` before showing the result of the form just evaluated. You can avoid this pause by setting `edebug-sit-on-break` to `nil`. See Section 19.2.16 [Edebug Options], page 365.

The `i` command steps into the function or macro called by the list form after point, and stops at its first stop point. Note that the form need not be the one about to be evaluated. But if the form is a function call about to be evaluated, remember to use this command before any of the arguments are evaluated, since otherwise it will be too late.

The `i` command instruments the function or macro it's supposed to step into, if it isn't instrumented already. This is convenient, but keep in mind that the function or macro remains instrumented unless you explicitly arrange to deinstrument it.

19.2.5 Miscellaneous Edebug Commands

Some miscellaneous Edebug commands are described here.

`?`
: Display the help message for Edebug (`edebug-help`).

`a`
`C-]`
: Abort one level back to the previous command level (`abort-recursive-edit`).

`q`
: Return to the top level editor command loop (`top-level`). This exits all recursive editing levels, including all levels of Edebug activity. However, instrumented code protected with `unwind-protect` or `condition-case` forms may resume debugging.

`Q`
: Like `q`, but don't stop even for protected code (`edebug-top-level-nonstop`).

`r`
: Redisplay the most recently known expression result in the echo area (`edebug-previous-result`).

`d`
: Display a backtrace, excluding Edebug's own functions for clarity (`edebug-pop-to-backtrace`).

  See Section 19.1.8 [Backtraces], page 339, for a description of backtraces and the commands which work on them.

  If you would like to see Edebug's functions in the backtrace, use *M-x edebug-backtrace-show-instrumentation*. To hide them again use *M-x edebug-backtrace-hide-instrumentation*.

  If a backtrace frame starts with '`>`' that means that Edebug knows where the source code for the frame is located. Use `s` to jump to the source code for the current frame.

  The backtrace buffer is killed automatically when you continue execution.

You can invoke commands from Edebug that activate Edebug again recursively. Whenever Edebug is active, you can quit to the top level with `q` or abort one recursive edit level with `C-]`. You can display a backtrace of all the pending evaluations with `d`.

19.2.6 Breaks

Edebug's step mode stops execution when the next stop point is reached. There are three other ways to stop Edebug execution once it has started: breakpoints, the global break condition, and source breakpoints.

19.2.6.1 Edebug Breakpoints

While using Edebug, you can specify *breakpoints* in the program you are testing: these are places where execution should stop. You can set a breakpoint at any stop point, as defined in Section 19.2.1 [Using Edebug], page 346. For setting and unsetting breakpoints, the stop point that is affected is the first one at or after point in the source code buffer. Here are the Edebug commands for breakpoints:

`b` Set a breakpoint at the stop point at or after point (`edebug-set-breakpoint`). If you use a prefix argument, the breakpoint is temporary—it turns off the first time it stops the program. An overlay with the `edebug-enabled-breakpoint` or `edebug-disabled-breakpoint` faces is put at the breakpoint.

`u` Unset the breakpoint (if any) at the stop point at or after point (`edebug-unset-breakpoint`).

`U` Unset any breakpoints in the current form (`edebug-unset-breakpoints`).

`D` Toggle whether to disable the breakpoint near point (`edebug-toggle-disable-breakpoint`). This command is mostly useful if the breakpoint is conditional and it would take some work to recreate the condition.

`x` *condition* `RET`
Set a conditional breakpoint which stops the program only if evaluating *condition* produces a non-`nil` value (`edebug-set-conditional-breakpoint`). With a prefix argument, the breakpoint is temporary.

`B` Move point to the next breakpoint in the current definition (`edebug-next-breakpoint`).

While in Edebug, you can set a breakpoint with `b` and unset one with `u`. First move point to the Edebug stop point of your choice, then type `b` or `u` to set or unset a breakpoint there. Unsetting a breakpoint where none has been set has no effect.

Re-evaluating or reinstrumenting a definition removes all of its previous breakpoints.

A *conditional breakpoint* tests a condition each time the program gets there. Any errors that occur as a result of evaluating the condition are ignored, as if the result were `nil`. To set a conditional breakpoint, use `x`, and specify the condition expression in the minibuffer. Setting a conditional breakpoint at a stop point that has a previously established conditional breakpoint puts the previous condition expression in the minibuffer so you can edit it.

You can make a conditional or unconditional breakpoint *temporary* by using a prefix argument with the command to set the breakpoint. When a temporary breakpoint stops the program, it is automatically unset.

Edebug always stops or pauses at a breakpoint, except when the Edebug mode is Gononstop. In that mode, it ignores breakpoints entirely.

To find out where your breakpoints are, use the `B` command, which moves point to the next breakpoint following point, within the same function, or to the first breakpoint if there

are no following breakpoints. This command does not continue execution—it just moves point in the buffer.

19.2.6.2 Global Break Condition

A *global break condition* stops execution when a specified condition is satisfied, no matter where that may occur. Edebug evaluates the global break condition at every stop point; if it evaluates to a non-`nil` value, then execution stops or pauses depending on the execution mode, as if a breakpoint had been hit. If evaluating the condition gets an error, execution does not stop.

The condition expression is stored in `edebug-global-break-condition`. You can specify a new expression using the `X` command from the source code buffer while Edebug is active, or using `C-x X X` from any buffer at any time, as long as Edebug is loaded (`edebug-set-global-break-condition`).

The global break condition is the simplest way to find where in your code some event occurs, but it makes code run much more slowly. So you should reset the condition to `nil` when not using it.

19.2.6.3 Source Breakpoints

All breakpoints in a definition are forgotten each time you reinstrument it. If you wish to make a breakpoint that won't be forgotten, you can write a *source breakpoint*, which is simply a call to the function `edebug` in your source code. You can, of course, make such a call conditional. For example, in the `fac` function, you can insert the first line as shown below, to stop when the argument reaches zero:

```
(defun fac (n)
  (if (= n 0) (edebug))
  (if (< 0 n)
      (* n (fac (1- n)))
    1))
```

When the `fac` definition is instrumented and the function is called, the call to `edebug` acts as a breakpoint. Depending on the execution mode, Edebug stops or pauses there.

If no instrumented code is being executed when `edebug` is called, that function calls `debug`.

19.2.7 Trapping Errors

Emacs normally displays an error message when an error is signaled and not handled with `condition-case`. While Edebug is active and executing instrumented code, it normally responds to all unhandled errors. You can customize this with the options `edebug-on-error` and `edebug-on-quit`; see Section 19.2.16 [Edebug Options], page 365.

When Edebug responds to an error, it shows the last stop point encountered before the error. This may be the location of a call to a function which was not instrumented, and within which the error actually occurred. For an unbound variable error, the last known stop point might be quite distant from the offending variable reference. In that case, you might want to display a full backtrace (see Section 19.2.5 [Edebug Misc], page 350).

If you change `debug-on-error` or `debug-on-quit` while Edebug is active, these changes will be forgotten when Edebug becomes inactive. Furthermore, during Edebug's recursive edit, these variables are bound to the values they had outside of Edebug.

19.2.8 Edebug Views

These Edebug commands let you view aspects of the buffer and window status as they were before entry to Edebug. The outside window configuration is the collection of windows and contents that were in effect outside of Edebug.

P

v Switch to viewing the outside window configuration (`edebug-view-outside`). Type *C-x X w* to return to Edebug.

p Temporarily display the outside current buffer with point at its outside position (`edebug-bounce-point`), pausing for one second before returning to Edebug. With a prefix argument *n*, pause for *n* seconds instead.

w Move point back to the current stop point in the source code buffer (`edebug-where`).

If you use this command in a different window displaying the same buffer, that window will be used instead to display the current definition in the future.

W Toggle whether Edebug saves and restores the outside window configuration (`edebug-toggle-save-windows`).

With a prefix argument, *W* only toggles saving and restoring of the selected window. To specify a window that is not displaying the source code buffer, you must use *C-x X W* from the global keymap.

You can view the outside window configuration with *v* or just bounce to the point in the current buffer with *p*, even if it is not normally displayed.

After moving point, you may wish to jump back to the stop point. You can do that with *w* from a source code buffer. You can jump back to the stop point in the source code buffer from any buffer using *C-x X w*.

Each time you use *W* to turn saving *off*, Edebug forgets the saved outside window configuration—so that even if you turn saving back *on*, the current window configuration remains unchanged when you next exit Edebug (by continuing the program). However, the automatic redisplay of `*edebug*` and `*edebug-trace*` may conflict with the buffers you wish to see unless you have enough windows open.

19.2.9 Evaluation

While within Edebug, you can evaluate expressions as if Edebug were not running. Edebug tries to be invisible to the expression's evaluation and printing. Evaluation of expressions that cause side effects will work as expected, except for changes to data that Edebug explicitly saves and restores. See Section 19.2.14 [The Outside Context], page 358, for details on this process.

e exp `RET` Evaluate expression *exp* in the context outside of Edebug (`edebug-eval-expression`). That is, Edebug tries to minimize its interference with the

evaluation. The result is shown in the echo area, or, if this command is given a prefix, pop up a new buffer and pretty-print the result there.

By default, this command suppresses the debugger during evaluation, so that an error in the evaluated expression won't add a new error on top of the existing one. Set the `debug-allow-recursive-debug` user option to a non-`nil` value to override this.

M-: exp RET
: Evaluate expression *exp* in the context of Edebug itself (`eval-expression`).

C-x C-e
: Evaluate the expression before point, in the context outside of Edebug (`edebug-eval-last-sexp`). With the prefix argument of zero (*C-u 0 C-x C-e*), don't shorten long items (like strings and lists). Any other prefix will result in the value being pretty-printed in a separate buffer.

Edebug supports evaluation of expressions containing references to lexically bound symbols created by the following constructs in `cl.el`: `lexical-let`, `macrolet`, and `symbol-macrolet`.

19.2.10 Evaluation List Buffer

You can use the *evaluation list buffer*, called `*edebug*`, to evaluate expressions interactively. You can also set up the *evaluation list* of expressions to be evaluated automatically each time Edebug updates the display.

E
: Switch to the evaluation list buffer `*edebug*` (`edebug-visit-eval-list`).

In the `*edebug*` buffer you can use the commands of Lisp Interaction mode (see Section "Lisp Interaction" in *The GNU Emacs Manual*) as well as these special commands:

C-j
: Evaluate the expression before point, in the outside context, and insert the value in the buffer (`edebug-eval-print-last-sexp`). With prefix argument of zero (*C-u 0 C-j*), don't shorten long items (like strings and lists).

C-x C-e
: Evaluate the expression before point, in the context outside of Edebug (`edebug-eval-last-sexp`).

C-c C-u
: Build a new evaluation list from the contents of the buffer (`edebug-update-eval-list`).

C-c C-d
: Delete the evaluation list group that point is in (`edebug-delete-eval-item`).

C-c C-w
: Switch back to the source code buffer at the current stop point (`edebug-where`).

You can evaluate expressions in the evaluation list window with *C-j* or *C-x C-e*, just as you would in `*scratch*`; but they are evaluated in the context outside of Edebug.

The expressions you enter interactively (and their results) are lost when you continue execution; but you can set up an *evaluation list* consisting of expressions to be evaluated each time execution stops.

To do this, write one or more *evaluation list groups* in the evaluation list buffer. An evaluation list group consists of one or more Lisp expressions. Groups are separated by comment lines.

The command `C-c C-u` (`edebug-update-eval-list`) rebuilds the evaluation list, scanning the buffer and using the first expression of each group. (The idea is that the second expression of the group is the value previously computed and displayed.)

Each entry to Edebug redisplays the evaluation list by inserting each expression in the buffer, followed by its current value. It also inserts comment lines so that each expression becomes its own group. Thus, if you type `C-c C-u` again without changing the buffer text, the evaluation list is effectively unchanged.

If an error occurs during an evaluation from the evaluation list, the error message is displayed in a string as if it were the result. Therefore, expressions using variables that are not currently valid do not interrupt your debugging.

Here is an example of what the evaluation list window looks like after several expressions have been added to it:

```
(current-buffer)
#<buffer *scratch*>
;-----------------------------------------------------------------
(selected-window)
#<window 16 on *scratch*>
;-----------------------------------------------------------------
(point)
196
;-----------------------------------------------------------------
bad-var
"Symbol's value as variable is void: bad-var"
;-----------------------------------------------------------------
(recursion-depth)
0
;-----------------------------------------------------------------
this-command
eval-last-sexp
;-----------------------------------------------------------------
```

To delete a group, move point into it and type `C-c C-d`, or simply delete the text for the group and update the evaluation list with `C-c C-u`. To add a new expression to the evaluation list, insert the expression at a suitable place, insert a new comment line, then type `C-c C-u`. You need not insert dashes in the comment line—its contents don't matter.

After selecting `*edebug*`, you can return to the source code buffer with `C-c C-w`. The `*edebug*` buffer is killed when you continue execution, and recreated next time it is needed.

19.2.11 Printing in Edebug

If an expression in your program produces a value containing circular list structure, you may get an error when Edebug attempts to print it.

One way to cope with circular structure is to set `print-length` or `print-level` to truncate the printing. Edebug does this for you; it binds `print-length` and `print-level` to the values of the variables `edebug-print-length` and `edebug-print-level` (so long as they have non-`nil` values). See Section 20.6 [Output Variables], page 381.

`edebug-print-length` [User Option]

If non-`nil`, Edebug binds `print-length` to this value while printing results. The default value is `50`.

`edebug-print-level` [User Option]
: If non-`nil`, Edebug binds `print-level` to this value while printing results. The default value is `50`.

You can also print circular structures and structures that share elements more informatively by binding `print-circle` to a non-`nil` value.

Here is an example of code that creates a circular structure:

```
(setq a (list 'x 'y))
(setcar a a)
```

If `print-circle` is non-`nil`, printing functions (e.g., `prin1`) will print `a` as '`#1=(#1# y)`'. The '`#1=`' notation labels the structure that follows it with the label '`1`', and the '`#1#`' notation references the previously labeled structure. This notation is used for any shared elements of lists or vectors.

`edebug-print-circle` [User Option]
: If non-`nil`, Edebug binds `print-circle` to this value while printing results. The default value is `t`.

For further details about how printing can be customized, see see Section 20.5 [Output Functions], page 378.

19.2.12 Trace Buffer

Edebug can record an execution trace, storing it in a buffer named `*edebug-trace*`. This is a log of function calls and returns, showing the function names and their arguments and values. To enable trace recording, set `edebug-trace` to a non-`nil` value.

Making a trace buffer is not the same thing as using trace execution mode (see Section 19.2.3 [Edebug Execution Modes], page 348).

When trace recording is enabled, each function entry and exit adds lines to the trace buffer. A function entry record consists of '`::::{`', followed by the function name and argument values. A function exit record consists of '`::::}`', followed by the function name and result of the function.

The number of '`:`'s in an entry shows its recursion depth. You can use the braces in the trace buffer to find the matching beginning or end of function calls.

You can customize trace recording for function entry and exit by redefining the functions `edebug-print-trace-before` and `edebug-print-trace-after`.

`edebug-tracing` *string body...* [Macro]
: This macro requests additional trace information around the execution of the *body* forms. The argument *string* specifies text to put in the trace buffer, after the '`{`' or '`}`'. All the arguments are evaluated, and `edebug-tracing` returns the value of the last form in *body*.

`edebug-trace` *format-string* **&rest** *format-args* [Function]
: This function inserts text in the trace buffer. It computes the text with `(apply 'format format-string format-args)`. It also appends a newline to separate entries.

`edebug-tracing` and `edebug-trace` insert lines in the trace buffer whenever they are called, even if Edebug is not active. Adding text to the trace buffer also scrolls its window to show the last lines inserted.

19.2.13 Coverage Testing

Edebug provides rudimentary coverage testing and display of execution frequency.

Coverage testing works by comparing the result of each expression with the previous result; each form in the program is considered covered if it has returned two different values since you began testing coverage in the current Emacs session. Thus, to do coverage testing on your program, execute it under various conditions and note whether it behaves correctly; Edebug will tell you when you have tried enough different conditions that each form has returned two different values.

Coverage testing makes execution slower, so it is only done if `edebug-test-coverage` is non-`nil`. Frequency counting is performed for all executions of an instrumented function, even if the execution mode is Go-nonstop, and regardless of whether coverage testing is enabled.

Use *C-x X =* (`edebug-display-freq-count`) to display both the coverage information and the frequency counts for a definition. Just `=` (`edebug-temp-display-freq-count`) displays the same information temporarily, only until you type another key.

`edebug-display-freq-count` [Command]

This command displays the frequency count data for each line of the current definition.

It inserts frequency counts as comment lines after each line of code. You can undo all insertions with one `undo` command. The counts appear under the '`(`' before an expression or the '`)`' after an expression, or on the last character of a variable. To simplify the display, a count is not shown if it is equal to the count of an earlier expression on the same line.

The character '`=`' following the count for an expression says that the expression has returned the same value each time it was evaluated. In other words, it is not yet covered for coverage testing purposes.

To clear the frequency count and coverage data for a definition, simply reinstrument it with `eval-defun`.

For example, after evaluating `(fac 5)` with a source breakpoint, and setting `edebug-test-coverage` to `t`, when the breakpoint is reached, the frequency data looks like this:

```
(defun fac (n)
  (if (= n 0) (edebug))
;#6           1      = =5
  (if (< 0 n)
;#5          =
      (* n (fac (1- n)))
;#    5               0
    1))
;#   0
```

The comment lines show that `fac` was called 6 times. The first `if` statement returned 5 times with the same result each time; the same is true of the condition on the second `if`. The recursive call of `fac` did not return at all.

19.2.14 The Outside Context

Edebug tries to be transparent to the program you are debugging, but it does not succeed completely. Edebug also tries to be transparent when you evaluate expressions with `e` or with the evaluation list buffer, by temporarily restoring the outside context. This section explains precisely what context Edebug restores, and how Edebug fails to be completely transparent.

19.2.14.1 Checking Whether to Stop

Whenever Edebug is entered, it needs to save and restore certain data before even deciding whether to make trace information or stop the program.

- `max-lisp-eval-depth` (see Section 10.4 [Eval], page 152) is increased to reduce Edebug's impact on the stack. You could, however, still run out of stack space when using Edebug. You can also enlarge the value of `edebug-max-depth` if Edebug reaches the limit of recursion depth instrumenting code that contains very large quoted lists.
- The state of keyboard macro execution is saved and restored. While Edebug is active, `executing-kbd-macro` is bound to `nil` unless `edebug-continue-kbd-macro` is non-`nil`.

19.2.14.2 Edebug Display Update

When Edebug needs to display something (e.g., in trace mode), it saves the current window configuration from outside Edebug (see Section 29.26 [Window Configurations], page 777). When you exit Edebug, it restores the previous window configuration.

Emacs redisplays only when it pauses. Usually, when you continue execution, the program re-enters Edebug at a breakpoint or after stepping, without pausing or reading input in between. In such cases, Emacs never gets a chance to redisplay the outside configuration. Consequently, what you see is the same window configuration as the last time Edebug was active, with no interruption.

Entry to Edebug for displaying something also saves and restores the following data (though some of them are deliberately not restored if an error or quit signal occurs).

- Which buffer is current, and the positions of point and the mark in the current buffer, are saved and restored.
- The outside window configuration is saved and restored if `edebug-save-windows` is non-`nil` (see Section 19.2.16 [Edebug Options], page 365). If the value of `edebug-save-windows` is a list, only the listed windows are saved and restored.

  The window configuration is not restored on error or quit, but the outside selected window *is* reselected even on error or quit in case a `save-excursion` is active.

  The window start and horizontal scrolling of the source code buffer are not restored, however, so that the display remains coherent within Edebug.

  Saving and restoring the outside window configuration can sometimes change the positions of point in the buffers on which the Lisp program you are debugging operates,

especially if your program moves point. If this happens and interferes with your debugging, we recommend to set `edebug-save-windows` to `nil` (see Section 19.2.16 [Edebug Options], page 365).

- The value of point in each displayed buffer is saved and restored if `edebug-save-displayed-buffer-points` is non-`nil`.
- The variables `overlay-arrow-position` and `overlay-arrow-string` are saved and restored, so you can safely invoke Edebug from the recursive edit elsewhere in the same buffer.
- `cursor-in-echo-area` is locally bound to `nil` so that the cursor shows up in the window.

19.2.14.3 Edebug Recursive Edit

When Edebug is entered and actually reads commands from the user, it saves (and later restores) these additional data:

- The current match data. See Section 35.6 [Match Data], page 1014.
- The variables `last-command`, `this-command`, `last-command-event`, `last-input-event`, `last-event-frame`, `last-nonmenu-event`, and `track-mouse`. Commands in Edebug do not affect these variables outside of Edebug.

  Executing commands within Edebug can change the key sequence that would be returned by `this-command-keys`, and there is no way to reset the key sequence from Lisp.

  Edebug cannot save and restore the value of `unread-command-events`. Entering Edebug while this variable has a nontrivial value can interfere with execution of the program you are debugging.
- Complex commands executed while in Edebug are added to the variable `command-history`. In rare cases this can alter execution.
- Within Edebug, the recursion depth appears one deeper than the recursion depth outside Edebug. This is not true of the automatically updated evaluation list window.
- `standard-output` and `standard-input` are bound to `nil` by the `recursive-edit`, but Edebug temporarily restores them during evaluations.
- The state of keyboard macro definition is saved and restored. While Edebug is active, `defining-kbd-macro` is bound to `edebug-continue-kbd-macro`.

19.2.15 Edebug and Macros

To make Edebug properly instrument expressions that call macros, some extra care is needed. This subsection explains the details.

19.2.15.1 Instrumenting Macro Calls

When Edebug instruments an expression that calls a Lisp macro, it needs additional information about the macro to do the job properly. This is because there is no a-priori way to tell which subexpressions of the macro call are forms to be evaluated. (Evaluation may occur explicitly in the macro body, or when the resulting expansion is evaluated, or any time later.)

Therefore, you must define an Edebug specification for each macro that Edebug will encounter, to explain the format of calls to that macro. To do this, add a `debug` declaration to the macro definition. Here is a simple example that shows the specification for the `for` example macro (see Section 14.5.2 [Argument Evaluation], page 272).

```
(defmacro for (var from init to final do &rest body)
  "Execute a simple \"for\" loop.
For example, (for i from 1 to 10 do (print i))."
  (declare (debug (symbolp "from" form "to" form "do" &rest form)))
  ...)
```

The Edebug specification says which parts of a call to the macro are forms to be evaluated. For simple macros, the specification often looks very similar to the formal argument list of the macro definition, but specifications are much more general than macro arguments. See Section 14.4 [Defining Macros], page 271, for more explanation of the `declare` form.

Take care to ensure that the specifications are known to Edebug when you instrument code. If you are instrumenting a function which uses a macro defined in another file, you may first need to either evaluate the `require` forms in the file containing your function, or explicitly load the file containing the macro. If the definition of a macro is wrapped by `eval-when-compile`, you may need to evaluate it.

You can also define an edebug specification for a macro separately from the macro definition with `def-edebug-spec`. Adding `debug` declarations is preferred, and more convenient, for macro definitions in Lisp, but `def-edebug-spec` makes it possible to define Edebug specifications for special forms implemented in C.

`def-edebug-spec` *macro specification* [Macro]
: Specify which expressions of a call to macro *macro* are forms to be evaluated. *specification* should be the Edebug specification. Neither argument is evaluated.

  The *macro* argument can actually be any symbol, not just a macro name.

Here is a table of the possibilities for *specification* and how each directs processing of arguments.

`t` All arguments are instrumented for evaluation. This is short for `(body)`.

a symbol The symbol must have an Edebug specification, which is used instead. This indirection is repeated until another kind of specification is found. This allows you to inherit the specification from another macro.

a list The elements of the list describe the types of the arguments of a calling form. The possible elements of a specification list are described in the following sections.

If a macro has no Edebug specification, neither through a `debug` declaration nor through a `def-edebug-spec` call, the variable `edebug-eval-macro-args` comes into play.

`edebug-eval-macro-args` [User Option]
: This controls the way Edebug treats macro arguments with no explicit Edebug specification. If it is `nil` (the default), none of the arguments is instrumented for evaluation. Otherwise, all arguments are instrumented.

19.2.15.2 Specification List

A *specification list* is required for an Edebug specification if some arguments of a macro call are evaluated while others are not. Some elements in a specification list match one or more arguments, but others modify the processing of all following elements. The latter, called *specification keywords*, are symbols beginning with '`&`' (such as `&optional`).

A specification list may contain sublists, which match arguments that are themselves lists, or it may contain vectors used for grouping. Sublists and groups thus subdivide the specification list into a hierarchy of levels. Specification keywords apply only to the remainder of the sublist or group they are contained in.

When a specification list involves alternatives or repetition, matching it against an actual macro call may require backtracking. For more details, see Section 19.2.15.3 [Backtracking], page 364.

Edebug specifications provide the power of regular expression matching, plus some context-free grammar constructs: the matching of sublists with balanced parentheses, recursive processing of forms, and recursion via indirect specifications.

Here's a table of the possible elements of a specification list, with their meanings (see Section 19.2.15.4 [Specification Examples], page 364, for the referenced examples):

`sexp` A single unevaluated Lisp object, which is not instrumented.

`form` A single evaluated expression, which is instrumented. If your macro wraps the expression with `lambda` before it is evaluated, use `def-form` instead. See `def-form` below.

`place` A generalized variable. See Section 12.17 [Generalized Variables], page 224.

`body` Short for `&rest form`. See `&rest` below. If your macro wraps its body of code with `lambda` before it is evaluated, use `def-body` instead. See `def-body` below.

`lambda-expr`
A lambda expression with no quoting.

`&optional`
All following elements in the specification list are optional; as soon as one does not match, Edebug stops matching at this level.

To make just a few elements optional, followed by non-optional elements, use `[&optional` *`specs`*`...]`. To specify that several elements must all match or none, use `&optional [`*`specs`*`...]`. See the `defun` example.

`&rest` All following elements in the specification list are repeated zero or more times. In the last repetition, however, it is not a problem if the expression runs out before matching all of the elements of the specification list.

To repeat only a few elements, use `[&rest` *`specs`*`...]`. To specify several elements that must all match on every repetition, use `&rest [`*`specs`*`...]`.

`&or` Each of the following elements in the specification list is an alternative. One of the alternatives must match, or the `&or` specification fails.

Each list element following `&or` is a single alternative. To group two or more list elements as a single alternative, enclose them in `[...]`.

`&not` Each of the following elements is matched as alternatives as if by using `&or`, but if any of them match, the specification fails. If none of them match, nothing is matched, but the `&not` specification succeeds.

`&define` Indicates that the specification is for a defining form. Edebug's definition of a defining form is a form containing one or more code forms which are saved and executed later, after the execution of the defining form.

The defining form itself is not instrumented (that is, Edebug does not stop before and after the defining form), but forms inside it typically will be instrumented. The `&define` keyword should be the first element in a list specification.

`nil` This is successful when there are no more arguments to match at the current argument list level; otherwise it fails. See sublist specifications and the backquote example.

`gate` No argument is matched but backtracking through the gate is disabled while matching the remainder of the specifications at this level. This is primarily used to generate more specific syntax error messages. See Section 19.2.15.3 [Backtracking], page 364, for more details. Also see the `let` example.

`&error` `&error` should be followed by a string, an error message, in the edebug-spec; it aborts the instrumentation, displaying the message in the minibuffer.

`&interpose`
Lets a function control the parsing of the remaining code. It takes the form `&interpose` *spec fun* `args...` and means that Edebug will first match *spec* against the code and then call *fun* with the code that matched `spec`, a parsing function *pf*, and finally *args...*. The parsing function expects a single argument indicating the specification list to use to parse the remaining code. It should be called exactly once and returns the instrumented code that *fun* is expected to return. For example `(&interpose symbolp pcase--match-pat-args)` matches sexps whose first element is a symbol and then lets `pcase--match-pat-args` lookup the specs associated with that head symbol according to `pcase--match-pat-args` and pass them to the *pf* it received as argument.

`other-symbol`
Any other symbol in a specification list may be a predicate or an indirect specification.

If the symbol has an Edebug specification, this *indirect specification* should be either a list specification that is used in place of the symbol, or a function that is called to process the arguments. The specification may be defined with `def-edebug-elem-spec`:

`def-edebug-elem-spec` *element specification* [Function]
Define the *specification* to use in place of the symbol *element*. *specification* has to be a list.

Otherwise, the symbol should be a predicate. The predicate is called with the argument, and if the predicate returns `nil`, the specification fails and the argument is not instrumented.

Some suitable predicates include `symbolp`, `integerp`, `stringp`, `vectorp`, and `atom`.

`[`*`elements`*`...]`
: A vector of elements groups the elements into a single *group specification*. Its meaning has nothing to do with vectors.

`"`*`string`*`"` The argument should be a symbol named *string*. This specification is equivalent to the quoted symbol, `'`*`symbol`*, where the name of *symbol* is the *string*, but the string form is preferred.

`(vector` *`elements`*`...)`
: The argument should be a vector whose elements must match the *elements* in the specification. See the backquote example.

`(`*`elements`*`...)`
: Any other list is a *sublist specification* and the argument must be a list whose elements match the specification *elements*.

  A sublist specification may be a dotted list and the corresponding list argument may then be a dotted list. Alternatively, the last CDR of a dotted list specification may be another sublist specification (via a grouping or an indirect specification, e.g., `(spec . [(more specs...)])`) whose elements match the non-dotted list arguments. This is useful in recursive specifications such as in the backquote example. Also see the description of a `nil` specification above for terminating such recursion.

  Note that a sublist specification written as `(specs . nil)` is equivalent to `(specs)`, and `(specs . (sublist-elements...))` is equivalent to `(specs sublist-elements...)`.

Here is a list of additional specifications that may appear only after `&define`. See the `defun` example.

`&name` Extracts the name of the current defining form from the code. It takes the form `&name [`*`prestring`*`]` *`spec`* `[`*`poststring`*`]` *`fun args...`* and means that Edebug will match *spec* against the code and then call *fun* with the concatenation of the current name, *args...*, *prestring*, the code that matched `spec`, and *poststring*. If *fun* is absent, it defaults to a function that concatenates the arguments (with an `@` between the previous name and the new).

`name` The argument, a symbol, is the name of the defining form. Shorthand for `[&name symbolp]`.

  A defining form is not required to have a name field; and it may have multiple name fields.

`arg` The argument, a symbol, is the name of an argument of the defining form. However, lambda-list keywords (symbols starting with '`&`') are not allowed.

`lambda-list`
: This matches a lambda list—the argument list of a lambda expression.

`def-body` The argument is the body of code in a definition. This is like `body`, described above, but a definition body must be instrumented with a different Edebug call

that looks up information associated with the definition. Use `def-body` for the highest level list of forms within the definition.

`def-form` The argument is a single, highest-level form in a definition. This is like `def-body`, except it is used to match a single form rather than a list of forms. As a special case, `def-form` also means that tracing information is not output when the form is executed. See the `interactive` example.

19.2.15.3 Backtracking in Specifications

If a specification fails to match at some point, this does not necessarily mean a syntax error will be signaled; instead, *backtracking* will take place until all alternatives have been exhausted. Eventually every element of the argument list must be matched by some element in the specification, and every required element in the specification must match some argument.

When a syntax error is detected, it might not be reported until much later, after higher-level alternatives have been exhausted, and with the point positioned further from the real error. But if backtracking is disabled when an error occurs, it can be reported immediately. Note that backtracking is also reenabled automatically in several situations; when a new alternative is established by `&optional`, `&rest`, or `&or`, or at the start of processing a sublist, group, or indirect specification. The effect of enabling or disabling backtracking is limited to the remainder of the level currently being processed and lower levels.

Backtracking is disabled while matching any of the form specifications (that is, `form`, `body`, `def-form`, and `def-body`). These specifications will match any form so any error must be in the form itself rather than at a higher level.

Backtracking is also disabled after successfully matching a quoted symbol, string specification, or `&define` keyword, since this usually indicates a recognized construct. But if you have a set of alternative constructs that all begin with the same symbol, you can usually work around this constraint by factoring the symbol out of the alternatives, e.g., `["foo" &or [first case] [second case] ...]`.

Most needs are satisfied by these two ways that backtracking is automatically disabled, but occasionally it is useful to explicitly disable backtracking by using the `gate` specification. This is useful when you know that no higher alternatives could apply. See the example of the `let` specification.

19.2.15.4 Specification Examples

It may be easier to understand Edebug specifications by studying the examples provided here.

Consider a hypothetical macro `my-test-generator` that runs tests on supplied lists of data. Although it is Edebug's default behavior to not instrument arguments as code, as controlled by `edebug-eval-macro-args` (see Section 19.2.15.1 [Instrumenting Macro Calls], page 359), it can be useful to explicitly document that the arguments are data:

```
(def-edebug-spec my-test-generator (&rest sexp))
```

A `let` special form has a sequence of bindings and a body. Each of the bindings is either a symbol or a sublist with a symbol and optional expression. In the specification below, notice the `gate` inside of the sublist to prevent backtracking once a sublist is found.

```
(def-edebug-spec let
```

```
((&rest
  &or symbolp (gate symbolp &optional form))
 body))
```

Edebug uses the following specifications for `defun` and the associated argument list and `interactive` specifications. It is necessary to handle interactive forms specially since an expression argument is actually evaluated outside of the function body. (The specification for `defmacro` is very similar to that for `defun`, but allows for the `declare` statement.)

```
(def-edebug-spec defun
  (&define name lambda-list
           [&optional stringp]   ; Match the doc string, if present.
           [&optional ("interactive" interactive)]
           def-body))

(def-edebug-elem-spec 'lambda-list
  '(([&rest arg]
     [&optional ["&optional" arg &rest arg]]
     &optional ["&rest" arg]
     )))

(def-edebug-elem-spec 'interactive
  '(&optional &or stringp def-form))    ; Notice: def-form
```

The specification for backquote below illustrates how to match dotted lists and use `nil` to terminate recursion. It also illustrates how components of a vector may be matched. (The actual specification defined by Edebug is a little different, and does not support dotted lists because doing so causes very deep recursion that could fail.)

```
(def-edebug-spec \` (backquote-form))   ; Alias just for clarity.

(def-edebug-elem-spec 'backquote-form
  '(&or ([&or "," ",@"] &or ("quote" backquote-form) form)
        (backquote-form . [&or nil backquote-form])
        (vector &rest backquote-form)
        sexp))
```

19.2.16 Edebug Options

These options affect the behavior of Edebug:

`edebug-setup-hook` [User Option]

Functions to call before Edebug is used. Each time it is set to a new value, Edebug will call those functions once and then reset `edebug-setup-hook` to `nil`. You could use this to load up Edebug specifications associated with a package you are using, but only when you also use Edebug. See Section 19.2.2 [Instrumenting], page 347.

`edebug-all-defs` [User Option]

If this is non-`nil`, normal evaluation of defining forms such as `defun` and `defmacro` instruments them for Edebug. This applies to `eval-defun`, `eval-region` and `eval-buffer`.

Use the command *M-x edebug-all-defs* to toggle the value of this option. See Section 19.2.2 [Instrumenting], page 347.

`edebug-all-forms` [User Option]

If this is non-`nil`, the commands `eval-defun`, `eval-region` and `eval-buffer` instrument all forms, even those that don't define anything. This doesn't apply to loading or evaluations in the minibuffer.

Use the command *M-x edebug-all-forms* to toggle the value of this option. See Section 19.2.2 [Instrumenting], page 347.

`edebug-eval-macro-args` [User Option]

When this is non-`nil`, all macro arguments will be instrumented in the generated code. For any macro, the `debug` declaration overrides this option. So to specify exceptions for macros that have some arguments evaluated and some not, use the `debug` declaration specify an Edebug form specification.

`edebug-save-windows` [User Option]

If this is non-`nil`, Edebug saves and restores the window configuration. That takes some time, so if your program does not care what happens to the window configurations, it is better to set this variable to `nil`. We also recommend to set this to `nil` if the default value causes Edebug to overwrite the positions of point in buffers that are involved in the program you are debugging, as result of saving and restoring the window configuration; this could happen if your program moves point in one or more of those buffers. Another option to try to customize in this case is `edebug-save-displayed-buffer-points`, described below.

If the value of `edebug-save-windows` is a list, only the listed windows are saved and restored.

You can use the *W* command in Edebug to change this variable interactively. See Section 19.2.14.2 [Edebug Display Update], page 358.

`edebug-save-displayed-buffer-points` [User Option]

If this is non-`nil`, Edebug saves and restores point in all displayed buffers.

Saving and restoring point in other buffers is necessary if you are debugging code that changes the point of a buffer that is displayed in a non-selected window. If Edebug or the user then selects the window, point in that buffer will move to the window's value of point.

Saving and restoring point in all buffers is expensive, since it requires selecting each window twice, so enable this only if you need it. See Section 19.2.14.2 [Edebug Display Update], page 358.

`edebug-initial-mode` [User Option]

If this variable is non-`nil`, it specifies the initial execution mode for Edebug when it is first activated. Possible values are `step`, `next`, `go`, `Go-nonstop`, `trace`, `Trace-fast`, `continue`, and `Continue-fast`.

The default value is `step`. This variable can be set interactively with *C-x C-a C-m* (`edebug-set-initial-mode`). See Section 19.2.3 [Edebug Execution Modes], page 348.

`edebug-trace` [User Option]
: If this is non-`nil`, trace each function entry and exit. Tracing output is displayed in a buffer named `*edebug-trace*`, one function entry or exit per line, indented by the recursion level.

  Also see `edebug-tracing`, in Section 19.2.12 [Trace Buffer], page 356.

`edebug-test-coverage` [User Option]
: If non-`nil`, Edebug tests coverage of all expressions debugged. See Section 19.2.13 [Coverage Testing], page 357.

`edebug-continue-kbd-macro` [User Option]
: If non-`nil`, continue defining or executing any keyboard macro that is executing outside of Edebug. Use this with caution since it is not debugged. See Section 19.2.3 [Edebug Execution Modes], page 348.

`edebug-print-length` [User Option]
: If non-`nil`, the default value of `print-length` for printing results in Edebug. See Section 20.6 [Output Variables], page 381.

`edebug-print-level` [User Option]
: If non-`nil`, the default value of `print-level` for printing results in Edebug. See Section 20.6 [Output Variables], page 381.

`edebug-print-circle` [User Option]
: If non-`nil`, the default value of `print-circle` for printing results in Edebug. See Section 20.6 [Output Variables], page 381.

`edebug-unwrap-results` [User Option]
: If non-`nil`, Edebug tries to remove any of its own instrumentation when showing the results of expressions. This is relevant when debugging macros where the results of expressions are themselves instrumented expressions. As a very artificial example, suppose that the example function `fac` has been instrumented, and consider a macro of the form:

```
(defmacro test () "Edebug example."
  (if (symbol-function 'fac)
      ...))
```

  If you instrument the `test` macro and step through it, then by default the result of the `symbol-function` call has numerous `edebug-after` and `edebug-before` forms, which can make it difficult to see the actual result. If `edebug-unwrap-results` is non-`nil`, Edebug tries to remove these forms from the result.

`edebug-on-error` [User Option]
: Edebug binds `debug-on-error` to this value, if `debug-on-error` was previously `nil`. See Section 19.2.7 [Trapping Errors], page 352.

`edebug-on-quit` [User Option]
: Edebug binds `debug-on-quit` to this value, if `debug-on-quit` was previously `nil`. See Section 19.2.7 [Trapping Errors], page 352.

If you change the values of `edebug-on-error` or `edebug-on-quit` while Edebug is active, their values won't be used until the *next* time Edebug is invoked via a new command.

`edebug-global-break-condition` [User Option]
: If non-`nil`, an expression to test for at every stop point. If the result is non-`nil`, then break. Errors are ignored. See Section 19.2.6.2 [Global Break Condition], page 352.

`edebug-sit-for-seconds` [User Option]
: Number of seconds to pause when a breakpoint is reached and the execution mode is trace or continue. See Section 19.2.3 [Edebug Execution Modes], page 348.

`edebug-sit-on-break` [User Option]
: Whether or not to pause for `edebug-sit-for-seconds` on reaching a breakpoint. Set to `nil` to prevent the pause, non-`nil` to allow it.

`edebug-behavior-alist` [User Option]
: By default, this alist contains one entry with the key `edebug` and a list of three functions, which are the default implementations of the functions inserted in instrumented code: `edebug-enter`, `edebug-before` and `edebug-after`. To change Edebug's behavior globally, modify the default entry.

  Edebug's behavior may also be changed on a per-definition basis by adding an entry to this alist, with a key of your choice and three functions. Then set the `edebug-behavior` symbol property of an instrumented definition to the key of the new entry, and Edebug will call the new functions in place of its own for that definition.

`edebug-new-definition-function` [User Option]
: A function run by Edebug after it wraps the body of a definition or closure. After Edebug has initialized its own data, this function is called with one argument, the symbol associated with the definition, which may be the actual symbol defined or one generated by Edebug. This function may be used to set the `edebug-behavior` symbol property of each definition instrumented by Edebug.

`edebug-after-instrumentation-function` [User Option]
: To inspect or modify Edebug's instrumentation before it is used, set this variable to a function which takes one argument, an instrumented top-level form, and returns either the same or a replacement form, which Edebug will then use as the final result of instrumentation.

19.3 Debugging Invalid Lisp Syntax

The Lisp reader reports invalid syntax, but cannot say where the real problem is. For example, the error '`End of file during parsing`' in evaluating an expression indicates an excess of open parentheses (or square brackets). The reader detects this imbalance at the end of the file, but it cannot figure out where the close parenthesis should have been. Likewise, '`Invalid read syntax: ")"`' indicates an excess close parenthesis or missing open parenthesis, but does not say where the missing parenthesis belongs. How, then, to find what to change?

If the problem is not simply an imbalance of parentheses, a useful technique is to try `C-M-e` (`end-of-defun`, see Section "Moving by Defuns" in *The GNU Emacs Manual*) at

the beginning of each defun, and see if it goes to the place where that defun appears to end. If it does not, there is a problem in that defun.

However, unmatched parentheses are the most common syntax errors in Lisp, and we can give further advice for those cases. (In addition, just moving point through the code with Show Paren mode enabled might find the mismatch.)

19.3.1 Excess Open Parentheses

The first step is to find the defun that is unbalanced. If there is an excess open parenthesis, the way to do this is to go to the end of the file and type `C-u C-M-u` (`backward-up-list`, see Section "Moving by Parens" in *The GNU Emacs Manual*). This will move you to the beginning of the first defun that is unbalanced.

The next step is to determine precisely what is wrong. There is no way to be sure of this except by studying the program, but often the existing indentation is a clue to where the parentheses should have been. The easiest way to use this clue is to reindent with `C-M-q` (`indent-pp-sexp`, see Section "Multi-line Indent" in *The GNU Emacs Manual*) and see what moves. **But don't do this yet!** Keep reading, first.

Before you do this, make sure the defun has enough close parentheses. Otherwise, `C-M-q` will get an error, or will reindent all the rest of the file until the end. So move to the end of the defun and insert a close parenthesis there. Don't use `C-M-e` (`end-of-defun`) to move there, since that too will fail to work until the defun is balanced.

Now you can go to the beginning of the defun and type `C-M-q`. Usually all the lines from a certain point to the end of the function will shift to the right. There is probably a missing close parenthesis, or a superfluous open parenthesis, near that point. (However, don't assume this is true; study the code to make sure.) Once you have found the discrepancy, undo the `C-M-q` with `C-_` (`undo`), since the old indentation is probably appropriate to the intended parentheses.

After you think you have fixed the problem, use `C-M-q` again. If the old indentation actually fit the intended nesting of parentheses, and you have put back those parentheses, `C-M-q` should not change anything.

19.3.2 Excess Close Parentheses

To deal with an excess close parenthesis, first go to the beginning of the file, then type `C-u -1 C-M-u` (`backward-up-list` with an argument of -1) to find the end of the first unbalanced defun.

Then find the actual matching close parenthesis by typing `C-M-f` (`forward-sexp`, see Section "Expressions" in *The GNU Emacs Manual*) at the beginning of that defun. This will leave you somewhere short of the place where the defun ought to end. It is possible that you will find a spurious close parenthesis in that vicinity.

If you don't see a problem at that point, the next thing to do is to type `C-M-q` (`indent-pp-sexp`) at the beginning of the defun. A range of lines will probably shift left; if so, the missing open parenthesis or spurious close parenthesis is probably near the first of those lines. (However, don't assume this is true; study the code to make sure.) Once you have found the discrepancy, undo the `C-M-q` with `C-_` (`undo`), since the old indentation is probably appropriate to the intended parentheses.

After you think you have fixed the problem, use *C-M-q* again. If the old indentation actually fits the intended nesting of parentheses, and you have put back those parentheses, *C-M-q* should not change anything.

19.4 Test Coverage

You can do coverage testing for a file of Lisp code by loading the `testcover` library and using the command *M-x testcover-start* `RET` *file* `RET` to instrument the code. Then test your code by calling it one or more times. Then use the command *M-x testcover-mark-all* to display colored highlights on the code to show where coverage is insufficient. The command *M-x testcover-next-mark* will move point forward to the next highlighted spot.

Normally, a red highlight indicates the form was never completely evaluated; a brown highlight means it always evaluated to the same value (meaning there has been little testing of what is done with the result). However, the red highlight is skipped for forms that can't possibly complete their evaluation, such as `error`. The brown highlight is skipped for forms that are expected to always evaluate to the same value, such as `(setq x 14)`.

For difficult cases, you can add do-nothing macros to your code to give advice to the test coverage tool.

`1value` *form* [Macro]
: Evaluate *form* and return its value, but inform coverage testing that *form*'s value should always be the same.

`noreturn` *form* [Macro]
: Evaluate *form*, informing coverage testing that *form* should never return. If it ever does return, you get a run-time error.

Edebug also has a coverage testing feature (see Section 19.2.13 [Coverage Testing], page 357). These features partly duplicate each other, and it would be cleaner to combine them.

19.5 Profiling

If your program is working correctly, but not fast enough, and you want to make it run more quickly or efficiently, the first thing to do is *profile* your code so that you know where it spends most of the execution time. If you find that one particular function is responsible for a significant portion of the execution time, you can start looking for ways to optimize that piece.

Emacs has built-in support for this. To begin profiling, type *M-x profiler-start*. You can choose to sample CPU usage periodically (`cpu`), when memory is allocated (`memory`), or both. Then run the code you'd like to speed up. After that, type *M-x profiler-report* to display a summary buffer for CPU usage sampled by each type (cpu and memory) that you chose to profile. The names of the report buffers include the times at which the reports were generated, so you can generate another report later on without erasing previous results. When you have finished profiling, type *M-x profiler-stop* (there is a small overhead associated with profiling, so we don't recommend leaving it active except when you are actually running the code you want to examine).

The profiler report buffer shows, on each line, a function that was called, preceded by how much CPU resources it used in absolute and percentage terms since profiling started. If a given line has a '+' symbol to the left of the function name, you can expand that line by typing `RET`, in order to see the function(s) called by the higher-level function. Use a prefix argument (*C-u* `RET`) to see the whole call tree below a function. Pressing `RET` again will collapse back to the original state.

Press *j* (`profiler-report-find-entry`) or *mouse-2* to jump to the definition of a function at point. Press *d* (`profiler-report-describe-entry`) to view a function's documentation. You can save a profile to a file using *C-x C-w* (`profiler-report-write-profile`) and read a saved profile with *M-x profiler-find-profile* or *M-x profiler-find-profile-other-window*. You can compare two profiles using `=` (`profiler-report-compare-profile`).

The `elp` library offers an alternative approach, which is useful when you know in advance which Lisp function(s) you want to profile. Using that library, you begin by setting `elp-function-list` to the list of function symbols—those are the functions you want to profile. Then type *M-x elp-instrument-list* `RET` *nil* `RET` to arrange for profiling those functions. After running the code you want to profile, invoke *M-x elp-results* to display the current results. See the file `elp.el` for more detailed instructions. This approach is limited to profiling functions written in Lisp, it cannot profile Emacs primitives.

You can measure the time it takes to evaluate individual Emacs Lisp forms using the `benchmark` library. See the function `benchmark-call` as well as the macros `benchmark-run`, `benchmark-run-compiled`, and `benchmark-progn` in `benchmark.el`. You can also use the `benchmark` command for timing forms interactively.

20 Reading and Printing Lisp Objects

Printing and *reading* are the operations of converting Lisp objects to textual form and vice versa. They use the printed representations and read syntax described in Chapter 2 [Lisp Data Types], page 8.

This chapter describes the Lisp functions for reading and printing. It also describes *streams*, which specify where to get the text (if reading) or where to put it (if printing).

20.1 Introduction to Reading and Printing

Reading a Lisp object means parsing a Lisp expression in textual form and producing a corresponding Lisp object. This is how Lisp programs get into Lisp from files of Lisp code. We call the text the *read syntax* of the object. For example, the text '`(a . 5)`' is the read syntax for a cons cell whose CAR is `a` and whose CDR is the number 5.

Printing a Lisp object means producing text that represents that object—converting the object to its *printed representation* (see Section 2.1 [Printed Representation], page 8). Printing the cons cell described above produces the text '`(a . 5)`'.

Reading and printing are more or less inverse operations: printing the object that results from reading a given piece of text often produces the same text, and reading the text that results from printing an object usually produces a similar-looking object. For example, printing the symbol `foo` produces the text '`foo`', and reading that text returns the symbol `foo`. Printing a list whose elements are `a` and `b` produces the text '`(a b)`', and reading that text produces a list (but not the same list) with elements `a` and `b`.

However, these two operations are not precisely inverse to each other. There are three kinds of exceptions:

- Printing can produce text that cannot be read. For example, buffers, windows, frames, subprocesses and markers print as text that starts with '`#`'; if you try to read this text, you get an error. There is no way to read those data types.
- One object can have multiple textual representations. For example, '`1`' and '`01`' represent the same integer, and '`(a b)`' and '`(a . (b))`' represent the same list. Reading will accept any of the alternatives, but printing must choose one of them.
- Comments can appear at certain points in the middle of an object's read sequence without affecting the result of reading it.

20.2 Input Streams

Most of the Lisp functions for reading text take an *input stream* as an argument. The input stream specifies where or how to get the characters of the text to be read. Here are the possible types of input stream:

buffer The input characters are read from *buffer*, starting with the character directly after point. Point advances as characters are read.

marker The input characters are read from the buffer that *marker* is in, starting with the character directly after the marker. The marker position advances as characters are read. The value of point in the buffer has no effect when the stream is a marker.

string The input characters are taken from *string*, starting at the first character in the string and using as many characters as required.

function The input characters are generated by *function*, which must support two kinds of calls:

- When it is called with no arguments, it should return the next character.
- When it is called with one argument (always a character), *function* should save the argument and arrange to return it on the next call. This is called *unreading* the character; it happens when the Lisp reader reads one character too many and wants to put it back where it came from. In this case, it makes no difference what value *function* returns.

`t` `t` used as a stream means that the input is read from the minibuffer. In fact, the minibuffer is invoked once and the text given by the user is made into a string that is then used as the input stream. If Emacs is running in batch mode (see Section 42.17 [Batch Mode], page 1294), standard input is used instead of the minibuffer. For example,

```
(message "%s" (read t))
```

will in batch mode read a Lisp expression from standard input and print the result to standard output.

`nil` `nil` supplied as an input stream means to use the value of `standard-input` instead; that value is the *default input stream*, and must be a non-`nil` input stream.

symbol A symbol as input stream is equivalent to the symbol's function definition (if any).

Here is an example of reading from a stream that is a buffer, showing where point is located before and after:

```
---------- Buffer: foo ----------
This⋆ is the contents of foo.
---------- Buffer: foo ----------

(read (get-buffer "foo"))
     ⇒ is
(read (get-buffer "foo"))
     ⇒ the

---------- Buffer: foo ----------
This is the⋆ contents of foo.
---------- Buffer: foo ----------
```

Note that the first read skips a space. Reading skips any amount of whitespace preceding the significant text.

Here is an example of reading from a stream that is a marker, initially positioned at the beginning of the buffer shown. The value read is the symbol `This`.

```
---------- Buffer: foo ----------
This is the contents of foo.
---------- Buffer: foo ----------

(setq m (set-marker (make-marker) 1 (get-buffer "foo")))
     ⇒ #<marker at 1 in foo>
(read m)
     ⇒ This
m
     ⇒ #<marker at 5 in foo>   ;; Before the first space.
```

Here we read from the contents of a string:

```
(read "(When in) the course")
     ⇒ (When in)
```

The following example reads from the minibuffer. The prompt is: ‘`Lisp expression: `’. (That is always the prompt used when you read from the stream `t`.) The user's input is shown following the prompt.

```
(read t)
     ⇒ 23
---------- Buffer: Minibuffer ----------
Lisp expression: 23 RET
---------- Buffer: Minibuffer ----------
```

Finally, here is an example of a stream that is a function, named `useless-stream`. Before we use the stream, we initialize the variable `useless-list` to a list of characters. Then each call to the function `useless-stream` obtains the next character in the list or unreads a character by adding it to the front of the list.

```
(setq useless-list (append "XY()" nil))
     ⇒ (88 89 40 41)

(defun useless-stream (&optional unread)
  (if unread
      (setq useless-list (cons unread useless-list))
    (prog1 (car useless-list)
           (setq useless-list (cdr useless-list)))))
     ⇒ useless-stream
```

Now we read using the stream thus constructed:

```
(read 'useless-stream)
     ⇒ XY

useless-list
     ⇒ (40 41)
```

Note that the open and close parentheses remain in the list. The Lisp reader encountered the open parenthesis, decided that it ended the input, and unread it. Another attempt to read from the stream at this point would read ‘`()`’ and return `nil`.

20.3 Input Functions

This section describes the Lisp functions and variables that pertain to reading.

In the functions below, *stream* stands for an input stream (see the previous section). If *stream* is `nil` or omitted, it defaults to the value of `standard-input`.

An `end-of-file` error is signaled if reading encounters an unterminated list, vector, or string.

`read` **&optional** *stream* [Function]

This function reads one textual Lisp expression from *stream*, returning it as a Lisp object. This is the basic Lisp input function.

`read-from-string` *string* **&optional** *start end* [Function]

This function reads the first textual Lisp expression from the text in *string*. It returns a cons cell whose CAR is that expression, and whose CDR is an integer giving the position of the next remaining character in the string (i.e., the first one not read).

If *start* is supplied, then reading begins at index *start* in the string (where the first character is at index 0). If you specify *end*, then reading is forced to stop just before that index, as if the rest of the string were not there.

For example:

```
(read-from-string "(setq x 55) (setq y 5)")
     ⇒ ((setq x 55) . 11)
(read-from-string "\"A short string\"")
     ⇒ ("A short string" . 16)

;; Read starting at the first character.
(read-from-string "(list 112)" 0)
     ⇒ ((list 112) . 10)
;; Read starting at the second character.
(read-from-string "(list 112)" 1)
     ⇒ (list . 5)
;; Read starting at the seventh character,
;;    and stopping at the ninth.
(read-from-string "(list 112)" 6 8)
     ⇒ (11 . 8)
```

`read-positioning-symbols` **&optional** *stream* [Function]

This function reads one textual expression from *stream*, like `read` does, but additionally positions the read symbols to the positions in *stream* where they occurred. Only the symbol `nil` is not positioned, this for efficiency reasons. See Section 9.6 [Symbols with Position], page 143. This function is used by the byte compiler.

`standard-input` [Variable]

This variable holds the default input stream—the stream that `read` uses when the *stream* argument is `nil`. The default is `t`, meaning use the minibuffer.

`read-circle` [Variable]

If non-`nil`, this variable enables the reading of circular and shared structures. See Section 2.6 [Circular Objects], page 30. Its default value is `t`.

When reading or writing from the standard input/output streams of the Emacs process in batch mode, it is sometimes required to make sure any arbitrary binary data will be read/written verbatim, and/or that no translation of newlines to or from CR-LF pairs is performed. This issue does not exist on POSIX hosts, only on MS-Windows and MS-DOS. The following function allows you to control the I/O mode of any standard stream of the Emacs process.

`set-binary-mode` *stream mode* [Function]
: Switch *stream* into binary or text I/O mode. If *mode* is non-`nil`, switch to binary mode, otherwise switch to text mode. The value of *stream* can be one of `stdin`, `stdout`, or `stderr`. This function flushes any pending output data of *stream* as a side effect, and returns the previous value of I/O mode for *stream*. On POSIX hosts, it always returns a non-`nil` value and does nothing except flushing pending output.

`readablep` *object* [Function]
: This predicate says whether *object* has *readable syntax*, i.e., it can be written out and then read back by the Emacs Lisp reader. If it can't, this function returns `nil`; if it can, this function returns a printed representation (via `prin1`, see Section 20.5 [Output Functions], page 378) of *object*.

20.4 Output Streams

An output stream specifies what to do with the characters produced by printing. Most print functions accept an output stream as an optional argument. Here are the possible types of output stream:

buffer The output characters are inserted into *buffer* at point. Point advances as characters are inserted.

marker The output characters are inserted into the buffer that *marker* points into, at the marker position. The marker position advances as characters are inserted. The value of point in the buffer has no effect on printing when the stream is a marker, and this kind of printing does not move point (except that if the marker points at or before the position of point, point advances with the surrounding text, as usual).

function The output characters are passed to *function*, which is responsible for storing them away. It is called with a single character as argument, as many times as there are characters to be output, and is responsible for storing the characters wherever you want to put them.

`t` The output characters are displayed in the echo area. If Emacs is running in batch mode (see Section 42.17 [Batch Mode], page 1294), the output is written to the standard output descriptor instead.

`nil` `nil` specified as an output stream means to use the value of the `standard-output` variable instead; that value is the *default output stream*, and must not be `nil`.

symbol A symbol as output stream is equivalent to the symbol's function definition (if any).

Many of the valid output streams are also valid as input streams. The difference between input and output streams is therefore more a matter of how you use a Lisp object, than of different types of object.

Here is an example of a buffer used as an output stream. Point is initially located as shown immediately before the '`h`' in '`the`'. At the end, point is located directly before that same '`h`'.

```
---------- Buffer: foo ----------
This is t*he contents of foo.
---------- Buffer: foo ----------

(print "This is the output" (get-buffer "foo"))
     ⇒ "This is the output"

---------- Buffer: foo ----------
This is t
"This is the output"
*he contents of foo.
---------- Buffer: foo ----------
```

Now we show a use of a marker as an output stream. Initially, the marker is in buffer `foo`, between the '`t`' and the '`h`' in the word '`the`'. At the end, the marker has advanced over the inserted text so that it remains positioned before the same '`h`'. Note that the location of point, shown in the usual fashion, has no effect.

```
---------- Buffer: foo ----------
This is the *output
---------- Buffer: foo ----------

(setq m (copy-marker 10))
     ⇒ #<marker at 10 in foo>

(print "More output for foo." m)
     ⇒ "More output for foo."

---------- Buffer: foo ----------
This is t
"More output for foo."
he *output
---------- Buffer: foo ----------

m
     ⇒ #<marker at 34 in foo>
```

The following example shows output to the echo area:

```
(print "Echo Area output" t)
     ⇒ "Echo Area output"
---------- Echo Area ----------
"Echo Area output"
---------- Echo Area ----------
```

Finally, we show the use of a function as an output stream. The function `eat-output` takes each character that it is given and conses it onto the front of the list `last-output` (see Section 5.4 [Building Lists], page 82). At the end, the list contains all the characters output, but in reverse order.

```
(setq last-output nil)
     ⇒ nil

(defun eat-output (c)
  (setq last-output (cons c last-output)))
     ⇒ eat-output

(print "This is the output" #'eat-output)
     ⇒ "This is the output"

last-output
     ⇒ (10 34 116 117 112 116 117 111 32 101 104
    116 32 115 105 32 115 105 104 84 34 10)
```

Now we can put the output in the proper order by reversing the list:

```
(concat (nreverse last-output))
     ⇒ "
\"This is the output\"
"
```

Calling `concat` converts the list to a string so you can see its contents more clearly.

`external-debugging-output` *character* [Function]

This function can be useful as an output stream when debugging. It writes *character* to the standard error stream.

For example

```
(print "This is the output" #'external-debugging-output)
⊣ This is the output
⇒ "This is the output"
```

20.5 Output Functions

This section describes the Lisp functions for printing Lisp objects—converting objects into their printed representation.

Some of the Emacs printing functions add quoting characters to the output when necessary so that it can be read properly. The quoting characters used are '`"`' and '`\`'; they distinguish strings from symbols, and prevent punctuation characters in strings and symbols from being taken as delimiters when reading. See Section 2.1 [Printed Representation], page 8, for full details. You specify quoting or no quoting by the choice of printing function.

If the text is to be read back into Lisp, then you should print with quoting characters to avoid ambiguity. Likewise, if the purpose is to describe a Lisp object clearly for a Lisp programmer. However, if the purpose of the output is to look nice for humans, then it is usually better to print without quoting.

Lisp objects can refer to themselves. Printing a self-referential object in the normal way would require an infinite amount of text, and the attempt could cause infinite recursion. Emacs detects such recursion and prints '`#level`' instead of recursively printing an object already being printed. For example, here '`#0`' indicates a recursive reference to the object at level 0 of the current print operation:

```
(setq foo (list nil))
     ⇒ (nil)
(setcar foo foo)
     ⇒ (#0)
```

In the functions below, *stream* stands for an output stream. (See the previous section for a description of output streams. Also See [external-debugging-output], page 378, a useful stream value for debugging.) If *stream* is `nil` or omitted, it defaults to the value of `standard-output`.

`print` *object* **&optional** *stream* [Function]

The `print` function is a convenient way of printing. It outputs the printed representation of *object* to *stream*, printing in addition one newline before *object* and another after it. Quoting characters are used. `print` returns *object*. For example:

```
(progn (print 'The\ cat\ in)
       (print "the hat")
       (print " came back"))
     ⊣
     ⊣ The\ cat\ in
     ⊣
     ⊣ "the hat"
     ⊣
     ⊣ " came back"
     ⇒ " came back"
```

`prin1` *object* **&optional** *stream overrides* [Function]

This function outputs the printed representation of *object* to *stream*. It does not print newlines to separate output as `print` does, but it does use quoting characters just like `print`. It returns *object*.

```
(progn (prin1 'The\ cat\ in)
       (prin1 "the hat")
       (prin1 " came back"))
     ⊣ The\ cat\ in"the hat"" came back"
     ⇒ " came back"
```

If *overrides* is non-`nil`, it should either be `t` (which tells `prin1` to use the defaults for all printer related variables), or a list of settings. See Section 20.7 [Output Overrides], page 384, for details.

`princ` *object* **&optional** *stream* [Function]

This function outputs the printed representation of *object* to *stream*. It returns *object*.

This function is intended to produce output that is readable by people, not by `read`, so it doesn't insert quoting characters and doesn't put double-quotes around the contents of strings. It does not add any spacing between calls.

```
(progn
  (princ 'The\ cat)
  (princ " in the \"hat\""))
     ⊣ The cat in the "hat"
     ⇒ " in the \"hat\""
```

`terpri` **&optional** *stream ensure* [Function]

This function outputs a newline to *stream*. The name stands for "terminate print". If *ensure* is non-`nil` no newline is printed if *stream* is already at the beginning of a line. Note in this case *stream* can not be a function and an error is signaled if it is. This function returns `t` if a newline is printed.

`write-char` *character* **&optional** *stream* [Function]

This function outputs *character* to *stream*. It returns *character*.

`flush-standard-output` [Function]

If you have Emacs-based batch scripts that send output to the terminal, Emacs will automatically display the output whenever you write a newline characters to `standard-output`. This function allows you to flush to `standard-output` without sending a newline character first, which enables you to display incomplete lines.

`prin1-to-string` *object* **&optional** *noescape overrides* [Function]

This function returns a string containing the text that `prin1` would have printed for the same argument.

```
(prin1-to-string 'foo)
     ⇒ "foo"
(prin1-to-string (mark-marker))
     ⇒ "#<marker at 2773 in strings.texi>"

If overrides is non-nil, it should either be t
(which tells prin1 to use the defaults for all printer related
variables), or a list of settings.  See Section 20.7 [Output Over-
rides], page 384, for details.
```

If *noescape* is non-`nil`, that inhibits use of quoting characters in the output. (This argument is supported in Emacs versions 19 and later.)

```
(prin1-to-string "foo")
     ⇒ "\"foo\""
(prin1-to-string "foo" t)
     ⇒ "foo"
```

See `format`, in Section 4.7 [Formatting Strings], page 66, for other ways to obtain the printed representation of a Lisp object as a string.

`with-output-to-string` *body...* [Macro]

This macro executes the *body* forms with `standard-output` set up to feed output into a string. Then it returns that string.

For example, if the current buffer name is '`foo`',

```
(with-output-to-string
  (princ "The buffer is ")
  (princ (buffer-name)))
```

returns `"The buffer is foo"`.

`pp` *object* **&optional** *stream* [Function]
: This function outputs *object* to *stream*, just like `prin1`, but does it in a prettier way. That is, it'll indent and fill the object to make it more readable for humans.

If you need to use binary I/O in batch mode, e.g., use the functions described in this section to write out arbitrary binary data or avoid conversion of newlines on non-POSIX hosts, see Section 20.3 [Input Functions], page 375.

20.6 Variables Affecting Output

`standard-output` [Variable]
: The value of this variable is the default output stream—the stream that print functions use when the *stream* argument is `nil`. The default is `t`, meaning display in the echo area.

`print-quoted` [Variable]
: If this is non-`nil`, that means to print quoted forms using abbreviated reader syntax, e.g., `(quote foo)` prints as `'foo`, and `(function foo)` as `#'foo`. The default is `t`.

`print-escape-newlines` [Variable]
: If this variable is non-`nil`, then newline characters in strings are printed as '`\n`' and formfeeds are printed as '`\f`'. Normally these characters are printed as actual newlines and formfeeds.

  This variable affects the print functions `prin1` and `print` that print with quoting. It does not affect `princ`. Here is an example using `prin1`:

```
(prin1 "a\nb")
     ⊣ "a
     ⊣ b"
     ⇒ "a
b"

(let ((print-escape-newlines t))
  (prin1 "a\nb"))
     ⊣ "a\nb"
     ⇒ "a
b"
```

  In the second expression, the local binding of `print-escape-newlines` is in effect during the call to `prin1`, but not during the printing of the result.

`print-escape-control-characters` [Variable]
: If this variable is non-`nil`, control characters in strings are printed as backslash sequences by the print functions `prin1` and `print` that print with quoting. If this

variable and `print-escape-newlines` are both non-`nil`, the latter takes precedences for newlines and formfeeds.

`print-escape-nonascii` [Variable]
If this variable is non-`nil`, then unibyte non-ASCII characters in strings are unconditionally printed as backslash sequences by the print functions `prin1` and `print` that print with quoting.

Those functions also use backslash sequences for unibyte non-ASCII characters, regardless of the value of this variable, when the output stream is a multibyte buffer or a marker pointing into one.

`print-escape-multibyte` [Variable]
If this variable is non-`nil`, then multibyte non-ASCII characters in strings are unconditionally printed as backslash sequences by the print functions `prin1` and `print` that print with quoting.

Those functions also use backslash sequences for multibyte non-ASCII characters, regardless of the value of this variable, when the output stream is a unibyte buffer or a marker pointing into one.

`print-charset-text-property` [Variable]
This variable controls printing of 'charset' text property on printing a string. The value should be `nil`, `t`, or `default`.

If the value is `nil`, `charset` text properties are never printed. If `t`, they are always printed.

If the value is `default`, only print `charset` text properties if there is an "unexpected" `charset` property. For ascii characters, all charsets are considered "expected". Otherwise, the expected `charset` property of a character is given by `char-charset`.

`print-length` [Variable]
The value of this variable is the maximum number of elements to print in any list, vector or bool-vector. If an object being printed has more than this many elements, it is abbreviated with an ellipsis.

If the value is `nil` (the default), then there is no limit.

```
(setq print-length 2)
     ⇒ 2
(print '(1 2 3 4 5))
     ⊣ (1 2 ...)
     ⇒ (1 2 ...)
```

`print-level` [Variable]
The value of this variable is the maximum depth of nesting of parentheses and brackets when printed. Any list or vector at a depth exceeding this limit is abbreviated with an ellipsis. A value of `nil` (which is the default) means no limit.

`eval-expression-print-length` [User Option]
`eval-expression-print-level` [User Option]
These are the values for `print-length` and `print-level` used by `eval-expression`, and thus, indirectly, by many interactive evaluation commands (see Section "Evaluating Emacs Lisp Expressions" in *The GNU Emacs Manual*).

These variables are used for detecting and reporting circular and shared structure:

`print-circle` [Variable]
: If non-`nil`, this variable enables detection of circular and shared structure in printing. See Section 2.6 [Circular Objects], page 30.

`print-unreadable-function` [Variable]
: By default, Emacs prints unreadable objects as '`#<...>`'. For instance:

```
(prin1-to-string (make-marker))
     ⇒ "#<marker in no buffer>"
```

: If this variable is non-`nil`, it should be a function that will be called to handle printing of these objects. The function will be called with two arguments: the object and the *noescape* flag used by the printing functions (see Section 20.5 [Output Functions], page 378).

: The function should return either `nil` (print the object as usual), or a string (which will be printed), or any other object (don't print the object). For instance:

```
(let ((print-unreadable-function
       (lambda (object escape) "hello")))
  (prin1-to-string (make-marker)))
     ⇒ "hello"
```

`print-gensym` [Variable]
: If non-`nil`, this variable enables detection of uninterned symbols (see Section 9.3 [Creating Symbols], page 135) in printing. When this is enabled, uninterned symbols print with the prefix '`#:`', which tells the Lisp reader to produce an uninterned symbol.

`print-continuous-numbering` [Variable]
: If non-`nil`, that means number continuously across print calls. This affects the numbers printed for '`#n=`' labels and '`#m#`' references. Don't set this variable with `setq`; you should only bind it temporarily to `t` with `let`. When you do that, you should also bind `print-number-table` to `nil`.

`print-number-table` [Variable]
: This variable holds a vector used internally by printing to implement the `print-circle` feature. You should not use it except to bind it to `nil` when you bind `print-continuous-numbering`.

`float-output-format` [Variable]
: This variable specifies how to print floating-point numbers. The default is `nil`, meaning use the shortest output that represents the number without losing information.

: To control output format more precisely, you can put a string in this variable. The string should hold a '`%`'-specification to be used in the C function `sprintf`. For further restrictions on what you can use, see the variable's documentation string.

`print-integers-as-characters` [Variable]
: When this variable is non-`nil`, integers that represent graphic base characters will be printed using Lisp character syntax (see Section 2.4.3.1 [Basic Char Syntax], page 11).

Other numbers are printed the usual way. For example, the list `(4 65 -1 10)` would be printed as '`(4 ?A -1 ?\n)`'.

More precisely, values printed in character syntax are those representing characters belonging to the Unicode general categories Letter, Number, Punctuation, Symbol and Private-use (see Section 34.6 [Character Properties], page 971), as well as the control characters having their own escape syntax such as newline.

20.7 Overriding Output Variables

The previous section (see Section 20.6 [Output Variables], page 381) lists the numerous variables that control how the Emacs Lisp printer formats data for outputs. These are generally available for users to change, but sometimes you want to output data in the default format, or override the user settings in some other way. For instance, if you're storing Emacs Lisp data in a file, you don't want that data to be shortened by a `print-length` setting.

The `prin1` and `prin1-to-string` functions therefore have an optional *overrides* argument. This argument can either be `t` (which means that all printing variables should be reset to the default values), or a list of settings for some of the variables. Each element in the list can be either `t` (which means "reset to defaults", and will usually be the first element of the list), or a pair whose `car` is a symbol that stands for an output variable and whose `cdr` is the value for that variable.

For instance, this prints using nothing but defaults:

```
(prin1 object nil t)
```

This prints *object* using the current printing settings, but overrides the value of `print-length` to be 5:

```
(prin1 object nil '((length . 5)))
```

And finally, this prints *object* using only default settings, but with `print-length` bound to 5:

```
(prin1 object nil '(t (length . 5)))
```

Below is a list of symbols that can be used, and which variables they map to:

`length` This overrides `print-length`.

`level` This overrides `print-level`.

`circle` This overrides `print-circle`.

`quoted` This overrides `print-quoted`.

`escape-newlines`
This overrides `print-escape-newlines`.

`escape-control-characters`
This overrides `print-escape-control-characters`.

`escape-nonascii`
This overrides `print-escape-nonascii`.

`escape-multibyte`
This overrides `print-escape-multibyte`.

`charset-text-property`
: This overrides `print-charset-text-property`.

`unreadeable-function`
: This overrides `print-unreadable-function`.

`gensym`
: This overrides `print-gensym`.

`continuous-numbering`
: This overrides `print-continuous-numbering`.

`number-table`
: This overrides `print-number-table`.

`float-format`
: This overrides `float-output-format`.

`integers-as-characters`
: This overrides `print-integers-as-characters`.

In the future, more overrides may be offered that do not map directly to a variable, but can only be used via this parameter.

21 Minibuffers

A *minibuffer* is a special buffer that Emacs commands use to read arguments more complicated than the single numeric prefix argument. These arguments include file names, buffer names, and command names (as in `M-x`). The minibuffer is displayed on the bottom line of the frame, in the same place as the echo area (see Section 41.4 [The Echo Area], page 1135), but only while it is in use for reading an argument.

21.1 Introduction to Minibuffers

In most ways, a minibuffer is a normal Emacs buffer. Most operations *within* a buffer, such as editing commands, work normally in a minibuffer. However, many operations for managing buffers do not apply to minibuffers. The name of a minibuffer always has the form ‘ `*Minibuf-`*`number`*`*`’, and it cannot be changed. Minibuffers are displayed only in special windows used only for minibuffers; these windows always appear at the bottom of a frame. (Sometimes frames have no minibuffer window, and sometimes a special kind of frame contains nothing but a minibuffer window; see Section 30.9 [Minibuffers and Frames], page 826.)

The text in the minibuffer always starts with the *prompt string*, the text that was specified by the program that is using the minibuffer to tell the user what sort of input to type. This text is marked read-only so you won't accidentally delete or change it. It is also marked as a field (see Section 33.19.9 [Fields], page 939), so that certain motion functions, including `beginning-of-line`, `forward-word`, `forward-sentence`, and `forward-paragraph`, stop at the boundary between the prompt and the actual text.

The minibuffer's window is normally a single line; it grows automatically if the contents require more space. Whilst the minibuffer is active, you can explicitly resize its window temporarily with the window sizing commands; the window reverts to its normal size when the minibuffer is exited. When the minibuffer is not active, you can resize its window permanently by using the window sizing commands in the frame's other window, or dragging the mode line with the mouse. (Due to details of the current implementation, for this to work `resize-mini-windows` must be `nil`.) If the frame contains just a minibuffer window, you can change its size by changing the frame's size.

Use of the minibuffer reads input events, and that alters the values of variables such as `this-command` and `last-command` (see Section 22.5 [Command Loop Info], page 437). Your program should bind them around the code that uses the minibuffer, if you do not want that to change them.

Under some circumstances, a command can use a minibuffer even if there is an active minibuffer; such a minibuffer is called a *recursive minibuffer*. The first minibuffer is named ‘ `*Minibuf-1*`’. Recursive minibuffers are named by incrementing the number at the end of the name. (The names begin with a space so that they won't show up in normal buffer lists.) Of several recursive minibuffers, the innermost (or most recently entered) is the *active minibuffer*–it is the one you can terminate by typing `RET` (`exit-minibuffer`) in. We usually call this *the* minibuffer. You can permit or forbid recursive minibuffers by setting the variable `enable-recursive-minibuffers`, or by putting properties of that name on command symbols (See Section 21.13 [Recursive Mini], page 421.)

Like other buffers, a minibuffer uses a local keymap (see Chapter 23 [Keymaps], page 481) to specify special key bindings. The function that invokes the minibuffer also sets up its local map according to the job to be done. See Section 21.2 [Text from Minibuffer], page 387, for the non-completion minibuffer local maps. See Section 21.6.3 [Completion Commands], page 401, for the minibuffer local maps for completion.

An active minibuffer usually has major mode `minibuffer-mode`. This is an Emacs internal mode without any special features. To customize the setup of minibuffers, we suggest you use `minibuffer-setup-hook` (see Section 21.15 [Minibuffer Misc], page 422) rather than `minibuffer-mode-hook`, since the former is run later, after the minibuffer has been fully initialized.

When a minibuffer is inactive, its major mode is `minibuffer-inactive-mode`, with keymap `minibuffer-inactive-mode-map`. This is only really useful if the minibuffer is in a separate frame. See Section 30.9 [Minibuffers and Frames], page 826.

When Emacs is running in batch mode, any request to read from the minibuffer actually reads a line from the standard input descriptor that was supplied when Emacs was started. This supports only basic input: none of the special minibuffer features (history, completion, etc.) are available in batch mode.

21.2 Reading Text Strings with the Minibuffer

The most basic primitive for minibuffer input is `read-from-minibuffer`, which can be used to read either a string or a Lisp object in textual form. The function `read-regexp` is used for reading regular expressions (see Section 35.3 [Regular Expressions], page 999), which are a special kind of string. There are also specialized functions for reading commands, variables, file names, etc. (see Section 21.6 [Completion], page 396).

In most cases, you should not call minibuffer input functions in the middle of a Lisp function. Instead, do all minibuffer input as part of reading the arguments for a command, in the `interactive` specification. See Section 22.2 [Defining Commands], page 425.

`read-from-minibuffer` *prompt* **&optional** *initial keymap read history default inherit-input-method* [Function]

This function is the most general way to get input from the minibuffer. By default, it accepts arbitrary text and returns it as a string; however, if *read* is non-`nil`, then it uses `read` to convert the text into a Lisp object (see Section 20.3 [Input Functions], page 375).

The first thing this function does is to activate a minibuffer and display it with *prompt* (which must be a string) as the prompt. Then the user can edit text in the minibuffer.

When the user types a command to exit the minibuffer, `read-from-minibuffer` constructs the return value from the text in the minibuffer. Normally it returns a string containing that text. However, if *read* is non-`nil`, `read-from-minibuffer` reads the text and returns the resulting Lisp object, unevaluated. (See Section 20.3 [Input Functions], page 375, for information about reading.)

The argument *default* specifies default values to make available through the history commands. It should be a string, a list of strings, or `nil`. The string or strings become the minibuffer's "future history", available to the user with `M-n`. In addition, if the call provides completion (e.g., via the *keymap* argument), the completion candidates

are added to the "future history" when the values in *default* are exhausted by `M-n`; see Section 21.4 [minibuffer-default-add-function], page 393.

If *read* is non-`nil`, then *default* is also used as the input to `read`, if the user enters empty input. If *default* is a list of strings, the first string is used as the input. If *default* is `nil`, empty input results in an `end-of-file` error. However, in the usual case (where *read* is `nil`), `read-from-minibuffer` ignores *default* when the user enters empty input and returns an empty string, `""`. In this respect, it differs from all the other minibuffer input functions in this chapter.

If *keymap* is non-`nil`, that keymap is the local keymap to use in the minibuffer. If *keymap* is omitted or `nil`, the value of `minibuffer-local-map` is used as the keymap. Specifying a keymap is the most important way to customize the minibuffer for various applications such as completion.

The argument *history* specifies a history list variable to use for saving the input and for history commands used in the minibuffer. It defaults to `minibuffer-history`. If *history* is the symbol `t`, history is not recorded. You can optionally specify a starting position in the history list as well. See Section 21.4 [Minibuffer History], page 393.

If the variable `minibuffer-allow-text-properties` is non-`nil`, then the string that is returned includes whatever text properties were present in the minibuffer. Otherwise all the text properties are stripped when the value is returned. (By default this variable is `nil`.)

The text properties in `minibuffer-prompt-properties` are applied to the prompt. By default, this property list defines a face to use for the prompt. This face, if present, is applied to the end of the face list and merged before display.

If the user wants to completely control the look of the prompt, the most convenient way to do that is to specify the `default` face at the end of all face lists. For instance:

```
(read-from-minibuffer
 (concat
  (propertize "Bold" 'face '(bold default))
  (propertize " and normal: " 'face '(default))))
```

If the argument *inherit-input-method* is non-`nil`, then the minibuffer inherits the current input method (see Section 34.11 [Input Methods], page 994) and the setting of `enable-multibyte-characters` (see Section 34.1 [Text Representations], page 966) from whichever buffer was current before entering the minibuffer.

Use of *initial* is mostly deprecated; we recommend using a non-`nil` value only in conjunction with specifying a cons cell for *history*. See Section 21.5 [Initial Input], page 395.

`read-string` *prompt* **&optional** *initial history default inherit-input-method* [Function]

This function reads a string from the minibuffer and returns it. The arguments *prompt*, *initial*, *history* and *inherit-input-method* are used as in `read-from-minibuffer`. The keymap used is `minibuffer-local-map`.

The optional argument *default* is used as in `read-from-minibuffer`, except that, if non-`nil`, it also specifies a default value to return if the user enters null input. As in `read-from-minibuffer` it should be a string, a list of strings, or `nil`, which is

equivalent to an empty string. When *default* is a string, that string is the default value. When it is a list of strings, the first string is the default value. (All these strings are available to the user in the "future minibuffer history".)

This function works by calling the `read-from-minibuffer` function:

```
(read-string prompt initial history default inherit)
≡
(let ((value
       (read-from-minibuffer prompt initial nil nil
                             history default inherit)))
  (if (and (equal value "") default)
      (if (consp default) (car default) default)
    value))
```

If you have a long string (for instance, one that is several lines long) that you wish to edit, using `read-string` may not be ideal. In that case, popping to a new, normal buffer where the user can edit the string may be more convenient, and you can use the `read-string-from-buffer` function to do that.

`read-regexp` *prompt* **&optional** *defaults history* [Function]

This function reads a regular expression as a string from the minibuffer and returns it. If the minibuffer prompt string *prompt* does not end in '`:`' (followed by optional whitespace), the function adds '`: `' to the end, preceded by the default return value (see below), if that is non-empty.

The optional argument *defaults* controls the default value to return if the user enters null input, and should be one of: a string; `nil`, which is equivalent to an empty string; a list of strings; or a symbol.

If *defaults* is a symbol, `read-regexp` consults the value of the variable `read-regexp-defaults-function` (see below), and if that is non-`nil` uses it in preference to *defaults*. The value in this case should be either:

- `regexp-history-last`, which means to use the first element of the appropriate minibuffer history list (see below).
- A function of no arguments, whose return value (which should be `nil`, a string, or a list of strings) becomes the value of *defaults*.

`read-regexp` now ensures that the result of processing *defaults* is a list (i.e., if the value is `nil` or a string, it converts it to a list of one element). To this list, `read-regexp` then appends a few potentially useful candidates for input. These are:

- The word or symbol at point.
- The last regexp used in an incremental search.
- The last string used in an incremental search.
- The last string or pattern used in query-replace commands.

The function now has a list of regular expressions that it passes to `read-from-minibuffer` to obtain the user's input. The first element of the list is the default result in case of empty input. All elements of the list are available to the user as the "future minibuffer history" list (see Section "Minibuffer History" in *The GNU Emacs Manual*).

The optional argument *history*, if non-`nil`, is a symbol specifying a minibuffer history list to use (see Section 21.4 [Minibuffer History], page 393). If it is omitted or `nil`, the history list defaults to `regexp-history`.

The user can use the *M-s c* command to indicate whether case folding should be on or off. If the user has used this command, the returned string will have the text property `case-fold` set to either `fold` or `inhibit-fold`. It is up to the caller of `read-regexp` to actually use this value, and the convenience function `read-regexp-case-fold-search` is provided for that. A typical usage pattern here might look like:

```
(let* ((regexp (read-regexp "Search for: "))
       (case-fold-search (read-regexp-case-fold-search regexp)))
  (re-search-forward regexp))
```

`read-regexp-defaults-function` [User Option]

The function `read-regexp` may use the value of this variable to determine its list of default regular expressions. If non-`nil`, the value of this variable should be either:

- The symbol `regexp-history-last`.
- A function of no arguments that returns either `nil`, a string, or a list of strings.

See `read-regexp` above for details of how these values are used.

`minibuffer-allow-text-properties` [Variable]

If this variable is `nil`, the default, then `read-from-minibuffer` and `read-string` strip all text properties from the minibuffer input before returning it. However, `read-no-blanks-input` (see below), as well as `read-minibuffer` and related functions (see Section 21.3 [Reading Lisp Objects With the Minibuffer], page 392), and all functions that do minibuffer input with completion, remove the `face` property unconditionally, regardless of the value of this variable.

If this variable is non-`nil`, most text properties on strings from the completion table are preserved—but only on the part of the strings that were completed.

```
(let ((minibuffer-allow-text-properties t))
  (completing-read "String: " (list (propertize "foobar" 'data 'zot))))
=> #("foobar" 3 6 (data zot))
```

In this example, the user typed 'foo' and then hit the *TAB* key, so the text properties are only preserved on the last three characters.

`minibuffer-local-map` [Variable]

This is the default local keymap for reading from the minibuffer. By default, it makes the following bindings:

C-j	`exit-minibuffer`
RET	`exit-minibuffer`
M-<	`minibuffer-beginning-of-buffer`
C-g	`abort-recursive-edit`
M-n DOWN	`next-history-element`

M-p UP	previous-history-element
M-s	next-matching-history-element
M-r	previous-matching-history-element

The variable `minibuffer-mode-map` is an alias for this variable.

`read-no-blanks-input` *prompt* **&optional** *initial* *inherit-input-method* [Function]

This function reads a string from the minibuffer, but does not allow whitespace characters as part of the input: instead, those characters terminate the input. The arguments *prompt*, *initial*, and *inherit-input-method* are used as in `read-from-minibuffer`.

This is a simplified interface to the `read-from-minibuffer` function, and passes the value of the `minibuffer-local-ns-map` keymap as the *keymap* argument for that function. Since the keymap `minibuffer-local-ns-map` does not rebind *C-q*, it *is* possible to put a space into the string, by quoting it.

This function discards text properties, regardless of the value of `minibuffer-allow-text-properties`.

```
(read-no-blanks-input prompt initial)
≡
(let (minibuffer-allow-text-properties)
  (read-from-minibuffer prompt initial minibuffer-local-ns-map))
```

`minibuffer-local-ns-map` [Variable]

This built-in variable is the keymap used as the minibuffer local keymap in the function `read-no-blanks-input`. By default, it makes the following bindings, in addition to those of `minibuffer-local-map`:

SPC	exit-minibuffer
TAB	exit-minibuffer
?	self-insert-and-exit

`format-prompt` *prompt default* **&rest** *format-args* [Function]

Format *prompt* with default value *default* according to the `minibuffer-default-prompt-format` variable.

`minibuffer-default-prompt-format` is a format string (defaulting to '`" (default %s)"`' that says how the "default" bit in prompts like '`"Local filename (default somefile): "`' are to be formatted.

To allow the users to customize how this is displayed, code that prompts the user for a value (and has a default) should look something along the lines of this code snippet:

```
(read-file-name
 (format-prompt "Local filename" file)
 nil file)
```

If *format-args* is `nil`, *prompt* is used as a literal string. If *format-args* is non-`nil`, *prompt* is used as a format control string, and *prompt* and *format-args* are passed to `format` (see Section 4.7 [Formatting Strings], page 66).

`minibuffer-default-prompt-format` can be '`""`', in which case no default values are displayed.

If *default* is `nil`, there is no default value, and therefore no "default value" string is included in the result value. If *default* is a non-`nil` list, the first element of the list is used in the prompt.

Both *prompt* and `minibuffer-default-prompt-format` are run through `substitute-command-keys` (see Section 25.3 [Keys in Documentation], page 600).

`read-minibuffer-restore-windows` [Variable]

If this option is non-`nil` (the default), getting input from the minibuffer will restore, on exit, the window configurations of the frame where the minibuffer was entered from and, if it is different, the frame that owns the minibuffer window. This means that if, for example, a user splits a window while getting input from the minibuffer on the same frame, that split will be undone when exiting the minibuffer.

If this option is `nil`, no such restorations are done. Hence, the window split mentioned above will persist after exiting the minibuffer.

21.3 Reading Lisp Objects with the Minibuffer

This section describes functions for reading Lisp objects with the minibuffer.

`read-minibuffer` *prompt* **&optional** *initial* [Function]

This function reads a Lisp object using the minibuffer, and returns it without evaluating it. The arguments *prompt* and *initial* are used as in `read-from-minibuffer`.

This is a simplified interface to the `read-from-minibuffer` function:

```
(read-minibuffer prompt initial)
≡
(let (minibuffer-allow-text-properties)
  (read-from-minibuffer prompt initial nil t))
```

Here is an example in which we supply the string `"(testing)"` as initial input:

```
(read-minibuffer
 "Enter an expression: " (format "%s" '(testing)))

;; Here is how the minibuffer is displayed:

---------- Buffer: Minibuffer ----------
Enter an expression: (testing)*
---------- Buffer: Minibuffer ----------
```

The user can type `RET` immediately to use the initial input as a default, or can edit the input.

`eval-minibuffer` *prompt* **&optional** *initial* [Function]

This function reads a Lisp expression using the minibuffer, evaluates it, then returns the result. The arguments *prompt* and *initial* are used as in `read-from-minibuffer`.

This function simply evaluates the result of a call to `read-minibuffer`:

```
(eval-minibuffer prompt initial)
≡
(eval (read-minibuffer prompt initial))
```

`edit-and-eval-command` *prompt form* [Function]

This function reads a Lisp expression in the minibuffer, evaluates it, then returns the result. The difference between this command and `eval-minibuffer` is that here the initial *form* is not optional and it is treated as a Lisp object to be converted to printed representation rather than as a string of text. It is printed with `prin1`, so if it is a string, double-quote characters (‘`"`’) appear in the initial text. See Section 20.5 [Output Functions], page 378.

In the following example, we offer the user an expression with initial text that is already a valid form:

```
(edit-and-eval-command "Please edit: " '(forward-word 1))

;; After evaluation of the preceding expression,
;;    the following appears in the minibuffer:

---------- Buffer: Minibuffer ----------
Please edit: (forward-word 1)⋆
---------- Buffer: Minibuffer ----------
```

Typing `RET` right away would exit the minibuffer and evaluate the expression, thus moving point forward one word.

21.4 Minibuffer History

A *minibuffer history list* records previous minibuffer inputs so the user can reuse them conveniently. It is a variable whose value is a list of strings (previous inputs), most recent first.

There are many separate minibuffer history lists, used for different kinds of inputs. It’s the Lisp programmer’s job to specify the right history list for each use of the minibuffer.

You specify a minibuffer history list with the optional *history* argument to `read-from-minibuffer` or `completing-read`. Here are the possible values for it:

variable Use *variable* (a symbol) as the history list.

(*variable* . *startpos*)
: Use *variable* (a symbol) as the history list, and assume that the initial history position is *startpos* (a nonnegative integer).

  Specifying 0 for *startpos* is equivalent to just specifying the symbol *variable*. `previous-history-element` will display the most recent element of the history list in the minibuffer. If you specify a positive *startpos*, the minibuffer history functions behave as if `(elt` *variable* `(1-` *startpos*`))` were the history element currently shown in the minibuffer.

  For consistency, you should also specify that element of the history as the initial minibuffer contents, using the *initial* argument to the minibuffer input function (see Section 21.5 [Initial Input], page 395).

If you don’t specify *history*, then the default history list `minibuffer-history` is used. For other standard history lists, see below. You can also create your own history list variable; just initialize it to `nil` before the first use. If the variable is buffer local, then each buffer will have its own input history list.

Both `read-from-minibuffer` and `completing-read` add new elements to the history list automatically, and provide commands to allow the user to reuse items on the list (see Section 21.10 [Minibuffer Commands], page 418). The only thing your program needs to do to use a history list is to initialize it and to pass its name to the input functions when you wish. But it is safe to modify the list by hand when the minibuffer input functions are not using it.

By default, when `M-n` (`next-history-element`, see Section 21.10 [next-history-element], page 418) reaches the end of the list of default values provided by the command which initiated reading input from the minibuffer, `M-n` adds all of the completion candidates, as specified by `minibuffer-completion-table` (see Section 21.6.3 [Completion Commands], page 401), to the list of defaults, so that all those candidates are available as "future history". Your program can control that via the variable `minibuffer-default-add-function`: if its value is not a function, this automatic addition is disabled, and you can also set this variable to your own function which adds only some candidates, or some other values, to the "future history".

Emacs functions that add a new element to a history list can also delete old elements if the list gets too long. The variable `history-length` specifies the maximum length for most history lists. To specify a different maximum length for a particular history list, put the length in the `history-length` property of the history list symbol. The variable `history-delete-duplicates` specifies whether to delete duplicates in history.

`add-to-history` *history-var newelt* **&optional** *maxelt keep-all* [Function]
: This function adds a new element *newelt*, if it isn't the empty string, to the history list stored in the variable *history-var*, and returns the updated history list. It limits the list length to the value of *maxelt* (if non-`nil`) or `history-length` (described below). The possible values of *maxelt* have the same meaning as the values of `history-length`. *history-var* cannot refer to a lexical variable.

  Normally, `add-to-history` removes duplicate members from the history list if `history-delete-duplicates` is non-`nil`. However, if *keep-all* is non-`nil`, that says not to remove duplicates, and to add *newelt* to the list even if it is empty.

`history-add-new-input` [Variable]
: If the value of this variable is `nil`, standard functions that read from the minibuffer don't add new elements to the history list. This lets Lisp programs explicitly manage input history by using `add-to-history`. The default value is `t`.

`history-length` [User Option]
: The value of this variable specifies the maximum length for all history lists that don't specify their own maximum lengths. If the value is `t`, that means there is no maximum (don't delete old elements). If a history list variable's symbol has a non-`nil` `history-length` property, it overrides this variable for that particular history list.

`history-delete-duplicates` [User Option]
: If the value of this variable is `t`, that means when adding a new history element, all previous identical elements are deleted.

Here are some of the standard minibuffer history list variables:

`minibuffer-history` [Variable]
: The default history list for minibuffer history input.

`query-replace-history` [Variable]
: A history list for arguments to `query-replace` (and similar arguments to other commands).

`file-name-history` [Variable]
: A history list for file-name arguments.

`buffer-name-history` [Variable]
: A history list for buffer-name arguments.

`regexp-history` [Variable]
: A history list for regular expression arguments.

`extended-command-history` [Variable]
: A history list for arguments that are names of extended commands.

`shell-command-history` [Variable]
: A history list for arguments that are shell commands.

`read-expression-history` [Variable]
: A history list for arguments that are Lisp expressions to evaluate.

`face-name-history` [Variable]
: A history list for arguments that are faces.

`custom-variable-history` [Variable]
: A history list for variable-name arguments read by `read-variable`.

`read-number-history` [Variable]
: A history list for numbers read by `read-number`.

`goto-line-history` [Variable]
: A history list for arguments to `goto-line`. This variable can be made local in every buffer by customizing the user option `goto-line-history-local`.

21.5 Initial Input

Several of the functions for minibuffer input have an argument called *initial*. This is a mostly-deprecated feature for specifying that the minibuffer should start out with certain text, instead of empty as usual.

If *initial* is a string, the minibuffer starts out containing the text of the string, with point at the end, when the user starts to edit the text. If the user simply types `RET` to exit the minibuffer, it will use the initial input string to determine the value to return.

We discourage use of a non-`nil` value for *initial*, because initial input is an intrusive interface. History lists and default values provide a much more convenient method to offer useful default inputs to the user.

There is just one situation where you should specify a string for an *initial* argument. This is when you specify a cons cell for the *history* argument. See Section 21.4 [Minibuffer History], page 393.

initial can also be a cons cell of the form `(`*`string`*` . `*`position`*`)`. This means to insert *string* in the minibuffer but put point at *position* within the string's text.

As a historical accident, *position* was implemented inconsistently in different functions. In `completing-read`, *position*'s value is interpreted as origin-zero; that is, a value of 0 means the beginning of the string, 1 means after the first character, etc. In `read-minibuffer`, and the other non-completion minibuffer input functions that support this argument, 1 means the beginning of the string, 2 means after the first character, etc.

Use of a cons cell as the value for *initial* arguments is deprecated.

21.6 Completion

Completion is a feature that fills in the rest of a name starting from an abbreviation for it. Completion works by comparing the user's input against a list of valid names and determining how much of the name is determined uniquely by what the user has typed. For example, when you type `C-x b` (`switch-to-buffer`) and then type the first few letters of the name of the buffer to which you wish to switch, and then type `TAB` (`minibuffer-complete`), Emacs extends the name as far as it can.

Standard Emacs commands offer completion for names of symbols, files, buffers, and processes; with the functions in this section, you can implement completion for other kinds of names.

The `try-completion` function is the basic primitive for completion: it returns the longest determined completion of a given initial string, with a given set of strings to match against.

The function `completing-read` provides a higher-level interface for completion. A call to `completing-read` specifies how to determine the list of valid names. The function then activates the minibuffer with a local keymap that binds a few keys to commands useful for completion. Other functions provide convenient simple interfaces for reading certain kinds of names with completion.

21.6.1 Basic Completion Functions

The following completion functions have nothing in themselves to do with minibuffers. We describe them here to keep them near the higher-level completion features that do use the minibuffer.

`try-completion` *string collection* **&optional** *predicate* [Function]

This function returns the longest common substring of all possible completions of *string* in *collection*.

collection is called the *completion table*. Its value must be a list of strings or cons cells, an obarray, a hash table, or a completion function.

`try-completion` compares *string* against each of the permissible completions specified by the completion table. If no permissible completions match, it returns `nil`. If there is just one matching completion, and the match is exact, it returns `t`. Otherwise, it returns the longest initial sequence common to all possible matching completions.

If *collection* is a list, the permissible completions are specified by the elements of the list, each of which should be either a string, or a cons cell whose CAR is either a string or a symbol (a symbol is converted to a string using `symbol-name`). If the list contains elements of any other type, those are ignored.

If *collection* is an obarray (see Section 9.3 [Creating Symbols], page 135), the names of all symbols in the obarray form the set of permissible completions.

If *collection* is a hash table, then the keys that are strings or symbols are the possible completions. Other keys are ignored.

You can also use a function as *collection*. Then the function is solely responsible for performing completion; `try-completion` returns whatever this function returns. The function is called with three arguments: *string*, *predicate* and `nil` (the third argument is so that the same function can be used in `all-completions` and do the appropriate thing in either case). See Section 21.6.7 [Programmed Completion], page 410.

If the argument *predicate* is non-`nil`, then it must be a function of one argument, unless *collection* is a hash table, in which case it should be a function of two arguments. It is used to test each possible match, and the match is accepted only if *predicate* returns non-`nil`. The argument given to *predicate* is either a string or a cons cell (the CAR of which is a string) from the alist, or a symbol (*not* a symbol name) from the obarray. If *collection* is a hash table, *predicate* is called with two arguments, the string key and the associated value.

In addition, to be acceptable, a completion must also match all the regular expressions in `completion-regexp-list`. (Unless *collection* is a function, in which case that function has to handle `completion-regexp-list` itself.)

In the first of the following examples, the string '`foo`' is matched by three of the alist CARs. All of the matches begin with the characters '`fooba`', so that is the result. In the second example, there is only one possible match, and it is exact, so the return value is `t`.

```
(try-completion
 "foo"
 '(("foobar1" 1) ("barfoo" 2) ("foobaz" 3) ("foobar2" 4)))
     ⇒ "fooba"

(try-completion "foo" '(("barfoo" 2) ("foo" 3)))
     ⇒ t
```

In the following example, numerous symbols begin with the characters '`forw`', and all of them begin with the word '`forward`'. In most of the symbols, this is followed with a '`-`', but not in all, so no more than '`forward`' can be completed.

```
(try-completion "forw" obarray)
     ⇒ "forward"
```

Finally, in the following example, only two of the three possible matches pass the predicate `test` (the string '`foobaz`' is too short). Both of those begin with the string '`foobar`'.

```
(defun test (s)
  (> (length (car s)) 6))
     ⇒ test
```

```
(try-completion
 "foo"
 '(("foobar1" 1) ("barfoo" 2) ("foobaz" 3) ("foobar2" 4))
 'test)
     ⇒ "foobar"
```

`all-completions` *string collection* **&optional** *predicate* [Function]

This function returns a list of all possible completions of *string*. The arguments to this function are the same as those of `try-completion`, and it uses `completion-regexp-list` in the same way that `try-completion` does.

If *collection* is a function, it is called with three arguments: *string*, *predicate* and `t`; then `all-completions` returns whatever the function returns. See Section 21.6.7 [Programmed Completion], page 410.

Here is an example, using the function `test` shown in the example for `try-completion`:

```
(defun test (s)
  (> (length (car s)) 6))
     ⇒ test

(all-completions
 "foo"
 '(("foobar1" 1) ("barfoo" 2) ("foobaz" 3) ("foobar2" 4))
 'test)
     ⇒ ("foobar1" "foobar2")
```

`test-completion` *string collection* **&optional** *predicate* [Function]

This function returns non-`nil` if *string* is a valid completion alternative specified by *collection* and *predicate*. The arguments are the same as in `try-completion`. For instance, if *collection* is a list of strings, this is true if *string* appears in the list and *predicate* is satisfied.

This function uses `completion-regexp-list` in the same way that `try-completion` does.

If *predicate* is non-`nil` and if *collection* contains several strings that are equal to each other, as determined by `compare-strings` according to `completion-ignore-case`, then *predicate* should accept either all or none of them. Otherwise, the return value of `test-completion` is essentially unpredictable.

If *collection* is a function, it is called with three arguments, the values *string*, *predicate* and `lambda`; whatever it returns, `test-completion` returns in turn.

`completion-boundaries` *string collection predicate suffix* [Function]

This function returns the boundaries of the field on which *collection* will operate, assuming that *string* holds the text before point and *suffix* holds the text after point.

Normally completion operates on the whole string, so for all normal collections, this will always return `(0 . (length` *`suffix`*`))`. But more complex completion, such as completion on files, is done one field at a time. For example, completion of `"/usr/sh"` will include `"/usr/share/"` but not `"/usr/share/doc"` even if `"/usr/share/doc"` exists. Also `all-completions` on `"/usr/sh"` will not include `"/usr/share/"` but only `"share/"`. So if *string* is `"/usr/sh"` and *suffix* is `"e/doc"`, `completion-boundaries` will return `(5 . 1)` which tells us that the *collection* will

only return completion information that pertains to the area after `"/usr/"` and before `"/doc"`. `try-completion` is not affected by nontrivial boundaries; e.g., `try-completion` on `"/usr/sh"` might still return `"/usr/share/"`, not `"share/"`.

If you store a completion alist in a variable, you should mark the variable as risky by giving it a non-`nil` `risky-local-variable` property. See Section 12.12 [File Local Variables], page 213.

`completion-ignore-case` [Variable]

If the value of this variable is non-`nil`, case is not considered significant in completion. Within `read-file-name`, this variable is overridden by `read-file-name-completion-ignore-case` (see Section 21.6.5 [Reading File Names], page 405); within `read-buffer`, it is overridden by `read-buffer-completion-ignore-case` (see Section 21.6.4 [High-Level Completion], page 403).

`completion-regexp-list` [Variable]

This is a list of regular expressions. The completion functions only consider a completion acceptable if it matches all regular expressions in this list, with `case-fold-search` (see Section 35.2 [Searching and Case], page 998) bound to the value of `completion-ignore-case`.

Do not set this variable to a non-`nil` value globally, as that is not safe and will probably cause errors in completion commands. This variable should be only let-bound to non-`nil` values around calls to basic completion functions: `try-completion`, `test-completion`, and `all-completions`.

`lazy-completion-table` *var fun* [Macro]

This macro provides a way to initialize the variable *var* as a collection for completion in a lazy way, not computing its actual contents until they are first needed. You use this macro to produce a value that you store in *var*. The actual computation of the proper value is done the first time you do completion using *var*. It is done by calling *fun* with no arguments. The value *fun* returns becomes the permanent value of *var*.

Here is an example:

```
(defvar foo (lazy-completion-table foo make-my-alist))
```

There are several functions that take an existing completion table and return a modified version. `completion-table-case-fold` returns a case-insensitive table. `completion-table-in-turn` and `completion-table-merge` combine multiple input tables in different ways. `completion-table-subvert` alters a table to use a different initial prefix. `completion-table-with-quoting` returns a table suitable for operating on quoted text. `completion-table-with-predicate` filters a table with a predicate function. `completion-table-with-terminator` adds a terminating string.

21.6.2 Completion and the Minibuffer

This section describes the basic interface for reading from the minibuffer with completion.

`completing-read` *prompt collection* **&optional** *predicate require-match initial history default inherit-input-method* [Function]

This function reads a string in the minibuffer, assisting the user by providing completion. It activates the minibuffer with prompt *prompt*, which must be a string.

The actual completion is done by passing the completion table *collection* and the completion predicate *predicate* to the function `try-completion` (see Section 21.6.1 [Basic Completion], page 396). This happens in certain commands bound in the local keymaps used for completion. Some of these commands also call `test-completion`. Thus, if *predicate* is non-`nil`, it should be compatible with *collection* and `completion-ignore-case`. See [Definition of test-completion], page 398.

See Section 21.6.7 [Programmed Completion], page 410, for detailed requirements when *collection* is a function.

The value of the optional argument *require-match* determines how the user may exit the minibuffer:

- If `nil`, the usual minibuffer exit commands work regardless of the input in the minibuffer.
- If `t`, the usual minibuffer exit commands won't exit unless the input completes to an element of *collection*.
- If `confirm`, the user can exit with any input, but is asked for confirmation if the input is not an element of *collection*.
- If `confirm-after-completion`, the user can exit with any input, but is asked for confirmation if the preceding command was a completion command (i.e., one of the commands in `minibuffer-confirm-exit-commands`) and the resulting input is not an element of *collection*. See Section 21.6.3 [Completion Commands], page 401.
- If a function, it is called with the input as the only argument. The function should return a non-`nil` value if the input is acceptable.
- Any other value of *require-match* behaves like `t`, except that the exit commands won't exit if it performs completion.

However, empty input is always permitted, regardless of the value of *require-match*; in that case, `completing-read` returns the first element of *default*, if it is a list; `""`, if *default* is `nil`; or *default*. The string or strings in *default* are also available to the user through the history commands (see Section 21.10 [Minibuffer Commands], page 418). In addition, the completion candidates are added to the "future history" when the values in *default* are exhausted by `M-n`; see Section 21.4 [minibuffer-default-add-function], page 393.

The function `completing-read` uses `minibuffer-local-completion-map` as the keymap if *require-match* is `nil`, and uses `minibuffer-local-must-match-map` if *require-match* is non-`nil`. See Section 21.6.3 [Completion Commands], page 401.

The argument *history* specifies which history list variable to use for saving the input and for minibuffer history commands. It defaults to `minibuffer-history`. If *history* is the symbol `t`, history is not recorded. See Section 21.4 [Minibuffer History], page 393.

The argument *initial* is mostly deprecated; we recommend using a non-`nil` value only in conjunction with specifying a cons cell for *history*. See Section 21.5 [Initial Input], page 395. For default input, use *default* instead.

If the argument *inherit-input-method* is non-`nil`, then the minibuffer inherits the current input method (see Section 34.11 [Input Methods], page 994) and the setting of

`enable-multibyte-characters` (see Section 34.1 [Text Representations], page 966) from whichever buffer was current before entering the minibuffer.

If the variable `completion-ignore-case` is non-`nil`, completion ignores case when comparing the input against the possible matches. See Section 21.6.1 [Basic Completion], page 396. In this mode of operation, *predicate* must also ignore case, or you will get surprising results.

Here's an example of using `completing-read`:

```
(completing-read
 "Complete a foo: "
 '(("foobar1" 1) ("barfoo" 2) ("foobaz" 3) ("foobar2" 4))
 nil t "fo")

;; After evaluation of the preceding expression,
;;    the following appears in the minibuffer:

---------- Buffer: Minibuffer ----------
Complete a foo: fo⋆
---------- Buffer: Minibuffer ----------
```

If the user then types `DEL DEL b RET`, `completing-read` returns `barfoo`.

The `completing-read` function binds variables to pass information to the commands that actually do completion. They are described in the following section.

`completing-read-function` [Variable]

The value of this variable must be a function, which is called by `completing-read` to actually do its work. It should accept the same arguments as `completing-read`. This can be bound to a different function to completely override the normal behavior of `completing-read`.

21.6.3 Minibuffer Commands that Do Completion

This section describes the keymaps, commands and user options used in the minibuffer to do completion.

`minibuffer-completion-table` [Variable]

The value of this variable is the completion table (see Section 21.6.1 [Basic Completion], page 396) used for completion in the minibuffer. This is the buffer-local variable that contains what `completing-read` passes to `try-completion`. It is used by minibuffer completion commands such as `minibuffer-complete`.

`minibuffer-completion-predicate` [Variable]

This variable's value is the predicate that `completing-read` passes to `try-completion`. The variable is also used by the other minibuffer completion functions.

`minibuffer-completion-confirm` [Variable]

This variable determines whether Emacs asks for confirmation before exiting the minibuffer; `completing-read` sets this variable, and the function `minibuffer-complete-and-exit` checks the value before exiting. If the value is `nil`, confirmation is not required. If the value is `confirm`, the user may exit with an input that is not a valid completion alternative, but Emacs asks for confirmation. If the value is

`confirm-after-completion`, the user may exit with an input that is not a valid completion alternative, but Emacs asks for confirmation if the user submitted the input right after any of the completion commands in `minibuffer-confirm-exit-commands`.

`minibuffer-confirm-exit-commands` [Variable]
This variable holds a list of commands that cause Emacs to ask for confirmation before exiting the minibuffer, if the *require-match* argument to `completing-read` is `confirm-after-completion`. The confirmation is requested if the user attempts to exit the minibuffer immediately after calling any command in this list.

`minibuffer-complete-word` [Command]
This function completes the minibuffer contents by at most a single word. Even if the minibuffer contents have only one completion, `minibuffer-complete-word` does not add any characters beyond the first character that is not a word constituent. See Chapter 36 [Syntax Tables], page 1025.

`minibuffer-complete` [Command]
This function completes the minibuffer contents as far as possible.

`minibuffer-complete-and-exit` [Command]
This function completes the minibuffer contents, and exits if confirmation is not required, i.e., if `minibuffer-completion-confirm` is `nil`. If confirmation *is* required, it is given by repeating this command immediately—the command is programmed to work without confirmation when run twice in succession.

`minibuffer-completion-help` [Command]
This function creates a list of the possible completions of the current minibuffer contents. It works by calling `all-completions` using the value of the variable `minibuffer-completion-table` as the *collection* argument, and the value of `minibuffer-completion-predicate` as the *predicate* argument. The list of completions is displayed as text in a buffer named `*Completions*`.

`display-completion-list` *completions* [Function]
This function displays *completions* to the stream in `standard-output`, usually a buffer. (See Chapter 20 [Read and Print], page 372, for more information about streams.) The argument *completions* is normally a list of completions just returned by `all-completions`, but it does not have to be. Each element may be a symbol or a string, either of which is simply printed. It can also be a list of two strings, which is printed as if the strings were concatenated. The first of the two strings is the actual completion, the second string serves as annotation.

This function is called by `minibuffer-completion-help`. A common way to use it is together with `with-output-to-temp-buffer`, like this:

```
(with-output-to-temp-buffer "*Completions*"
  (display-completion-list
    (all-completions (buffer-string) my-alist)))
```

`completion-auto-help` [User Option]
: If this variable is non-`nil`, the completion commands automatically display a list of possible completions whenever nothing can be completed because the next character is not uniquely determined.

`minibuffer-local-completion-map` [Variable]
: `completing-read` uses this value as the local keymap when an exact match of one of the completions is not required. By default, this keymap makes the following bindings:

?	`minibuffer-completion-help`
SPC	`minibuffer-complete-word`
TAB	`minibuffer-complete`

: and uses `minibuffer-local-map` as its parent keymap (see [Definition of minibuffer-local-map], page 390).

`minibuffer-local-must-match-map` [Variable]
: `completing-read` uses this value as the local keymap when an exact match of one of the completions is required. Therefore, no keys are bound to `exit-minibuffer`, the command that exits the minibuffer unconditionally. By default, this keymap makes the following bindings:

C-j	`minibuffer-complete-and-exit`
RET	`minibuffer-complete-and-exit`

: and uses `minibuffer-local-completion-map` as its parent keymap.

`minibuffer-local-filename-completion-map` [Variable]
: This is a sparse keymap that simply unbinds `SPC`; because filenames can contain spaces. The function `read-file-name` combines this keymap with either `minibuffer-local-completion-map` or `minibuffer-local-must-match-map`.

`minibuffer-beginning-of-buffer-movement` [Variable]
: If non-`nil`, the *M-<* command will move to the end of the prompt if point is after the end of the prompt. If point is at or before the end of the prompt, move to the start of the buffer. If this variable is `nil`, the command behaves like `beginning-of-buffer`.

21.6.4 High-Level Completion Functions

This section describes the higher-level convenience functions for reading certain sorts of names with completion.

In most cases, you should not call these functions in the middle of a Lisp function. When possible, do all minibuffer input as part of reading the arguments for a command, in the `interactive` specification. See Section 22.2 [Defining Commands], page 425.

`read-buffer` *prompt* **&optional** *default require-match predicate* [Function]
: This function reads the name of a buffer and returns it as a string. It prompts with *prompt*. The argument *default* is the default name to use, the value to return if the user exits with an empty minibuffer. If non-`nil`, it should be a string, a list of

strings, or a buffer. If it is a list, the default value is the first element of this list. It is mentioned in the prompt, but is not inserted in the minibuffer as initial input.

The argument *prompt* should be a string ending with a colon and a space. If *default* is non-`nil`, the function inserts it in *prompt* before the colon to follow the convention for reading from the minibuffer with a default value (see Section D.3 [Programming Tips], page 1340).

The optional argument *require-match* has the same meaning as in `completing-read`. See Section 21.6.2 [Minibuffer Completion], page 399.

The optional argument *predicate*, if non-`nil`, specifies a function to filter the buffers that should be considered: the function will be called with every potential candidate as its argument, and should return `nil` to reject the candidate, non-`nil` to accept it.

In the following example, the user enters '`minibuffer.t`', and then types `RET`. The argument *require-match* is `t`, and the only buffer name starting with the given input is '`minibuffer.texi`', so that name is the value.

```
(read-buffer "Buffer name: " "foo" t)
;; After evaluation of the preceding expression,
;;    the following prompt appears,
;;    with an empty minibuffer:

---------- Buffer: Minibuffer ----------
Buffer name (default foo): ⋆
---------- Buffer: Minibuffer ----------

;; The user types minibuffer.t RET.
     ⇒ "minibuffer.texi"
```

`read-buffer-function` [User Option]

This variable, if non-`nil`, specifies a function for reading buffer names. `read-buffer` calls this function instead of doing its usual work, with the same arguments passed to `read-buffer`.

`read-buffer-completion-ignore-case` [User Option]

If this variable is non-`nil`, `read-buffer` ignores case when performing completion while reading the buffer name.

`read-command` *prompt* **&optional** *default* [Function]

This function reads the name of a command and returns it as a Lisp symbol. The argument *prompt* is used as in `read-from-minibuffer`. Recall that a command is anything for which `commandp` returns `t`, and a command name is a symbol for which `commandp` returns `t`. See Section 22.3 [Interactive Call], page 433.

The argument *default* specifies what to return if the user enters null input. It can be a symbol, a string or a list of strings. If it is a string, `read-command` interns it before returning it. If it is a list, `read-command` interns the first element of this list. If *default* is `nil`, that means no default has been specified; then if the user enters null input, the return value is `(intern "")`, that is, a symbol whose name is an empty string, and whose printed representation is `##` (see Section 2.4.4 [Symbol Type], page 14).

```
(read-command "Command name? ")
```

```
;; After evaluation of the preceding expression,
;;    the following prompt appears with an empty minibuffer:

---------- Buffer: Minibuffer ----------
Command name?
---------- Buffer: Minibuffer ----------
```

If the user types *forward-c* `RET`, then this function returns `forward-char`.

The `read-command` function is a simplified interface to `completing-read`. It uses the variable `obarray` so as to complete in the set of extant Lisp symbols, and it uses the `commandp` predicate so as to accept only command names:

```
(read-command prompt)
≡
(intern (completing-read prompt obarray
                         'commandp t nil))
```

`read-variable` *prompt* **&optional** *default* [Function]

This function reads the name of a customizable variable and returns it as a symbol. Its arguments have the same form as those of `read-command`. It behaves just like `read-command`, except that it uses the predicate `custom-variable-p` instead of `commandp`.

`read-color` **&optional** *prompt convert allow-empty display* [Command]

This function reads a string that is a color specification, either the color's name or an RGB hex value such as `#RRRGGGBBB`. It prompts with *prompt* (default: `"Color (name or #RGB triplet):"`) and provides completion for color names, but not for hex RGB values. In addition to names of standard colors, completion candidates include the foreground and background colors at point.

Valid RGB values are described in Section 30.23 [Color Names], page 848.

The function's return value is the string typed by the user in the minibuffer. However, when called interactively or if the optional argument *convert* is non-`nil`, it converts any input color name into the corresponding RGB value string and instead returns that. This function requires a valid color specification to be input. Empty color names are allowed when *allow-empty* is non-`nil` and the user enters null input.

Interactively, or when *display* is non-`nil`, the return value is also displayed in the echo area.

See also the functions `read-coding-system` and `read-non-nil-coding-system`, in Section 34.10.4 [User-Chosen Coding Systems], page 985, and `read-input-method-name`, in Section 34.11 [Input Methods], page 994.

21.6.5 Reading File Names

The high-level completion functions `read-file-name`, `read-directory-name`, and `read-shell-command` are designed to read file names, directory names, and shell commands, respectively. They provide special features, including automatic insertion of the default directory.

`read-file-name` *prompt* **&optional** *directory default require-match initial predicate* [Function]

This function reads a file name, prompting with *prompt* and providing completion.

As an exception, this function reads a file name using a graphical file dialog instead of the minibuffer, if all of the following are true:

1. It is invoked via a mouse command.
2. The selected frame is on a graphical display supporting such dialogs.
3. The variable `use-dialog-box` is non-`nil`. See Section "Dialog Boxes" in *The GNU Emacs Manual*.
4. The *directory* argument, described below, does not specify a remote file. See Section "Remote Files" in *The GNU Emacs Manual*.

The exact behavior when using a graphical file dialog is platform-dependent. Here, we simply document the behavior when using the minibuffer.

`read-file-name` does not automatically expand the returned file name. You can call `expand-file-name` yourself if an absolute file name is required.

The optional argument *require-match* has the same meaning as in `completing-read`. See Section 21.6.2 [Minibuffer Completion], page 399.

The argument *directory* specifies the directory to use for completing relative file names. It should be an absolute directory name. If the variable `insert-default-directory` is non-`nil`, *directory* is also inserted in the minibuffer as initial input. It defaults to the current buffer's value of `default-directory`.

If you specify *initial*, that is an initial file name to insert in the buffer (after *directory*, if that is inserted). In this case, point goes at the beginning of *initial*. The default for *initial* is `nil`—don't insert any file name. To see what *initial* does, try the command `C-x C-v` in a buffer visiting a file. **Please note:** we recommend using *default* rather than *initial* in most cases.

If *default* is non-`nil`, then the function returns *default* if the user exits the minibuffer with the same non-empty contents that `read-file-name` inserted initially. The initial minibuffer contents are always non-empty if `insert-default-directory` is non-`nil`, as it is by default. *default* is not checked for validity, regardless of the value of *require-match*. However, if *require-match* is non-`nil`, the initial minibuffer contents should be a valid file (or directory) name. Otherwise `read-file-name` attempts completion if the user exits without any editing, and does not return *default*. *default* is also available through the history commands.

If *default* is `nil`, `read-file-name` tries to find a substitute default to use in its place, which it treats in exactly the same way as if it had been specified explicitly. If *default* is `nil`, but *initial* is non-`nil`, then the default is the absolute file name obtained from *directory* and *initial*. If both *default* and *initial* are `nil` and the buffer is visiting a file, `read-file-name` uses the absolute file name of that file as default. If the buffer is not visiting a file, then there is no default. In that case, if the user types `RET` without any editing, `read-file-name` simply returns the pre-inserted contents of the minibuffer.

If the user types `RET` in an empty minibuffer, this function returns an empty string, regardless of the value of *require-match*. This is, for instance, how the user can make the current buffer visit no file using *`M-x set-visited-file-name`*.

If *predicate* is non-`nil`, it specifies a function of one argument that decides which file names are acceptable completion alternatives. A file name is an acceptable value if *predicate* returns non-`nil` for it.

Here is an example of using `read-file-name`:

```
(read-file-name "The file is ")

;; After evaluation of the preceding expression,
;;    the following appears in the minibuffer:

---------- Buffer: Minibuffer ----------
The file is /gp/gnu/elisp/⋆
---------- Buffer: Minibuffer ----------
```

Typing *`manual`* `TAB` results in the following:

```
---------- Buffer: Minibuffer ----------
The file is /gp/gnu/elisp/manual.texi⋆
---------- Buffer: Minibuffer ----------
```

If the user types `RET`, `read-file-name` returns the file name as the string `"/gp/gnu/elisp/manual.texi"`.

`read-file-name-function` [Variable]

If non-`nil`, this should be a function that accepts the same arguments as `read-file-name`. When `read-file-name` is called, it calls this function with the supplied arguments instead of doing its usual work.

`read-file-name-completion-ignore-case` [User Option]

If this variable is non-`nil`, `read-file-name` ignores case when performing completion.

`read-directory-name` *prompt* **&optional** *directory default require-match initial* [Function]

This function is like `read-file-name` but allows only directory names as completion alternatives.

If *default* is `nil` and *initial* is non-`nil`, `read-directory-name` constructs a substitute default by combining *directory* (or the current buffer's default directory if *directory* is `nil`) and *initial*. If both *default* and *initial* are `nil`, this function uses *directory* as substitute default, or the current buffer's default directory if *directory* is `nil`.

`insert-default-directory` [User Option]

This variable is used by `read-file-name`, and thus, indirectly, by most commands reading file names. (This includes all commands that use the code letters '`f`' or '`F`' in their interactive form. See Section 22.2.2 [Code Characters for interactive], page 428.) Its value controls whether `read-file-name` starts by placing the name of the default directory in the minibuffer, plus the initial file name, if any. If the value of this variable is `nil`, then `read-file-name` does not place any initial input in the minibuffer (unless you specify initial input with the *initial* argument). In that

case, the default directory is still used for completion of relative file names, but is not displayed.

If this variable is `nil` and the initial minibuffer contents are empty, the user may have to explicitly fetch the next history element to access a default value. If the variable is non-`nil`, the initial minibuffer contents are always non-empty and the user can always request a default value by immediately typing `RET` in an unedited minibuffer. (See above.)

For example:

```
;; Here the minibuffer starts out with the default directory.
(let ((insert-default-directory t))
  (read-file-name "The file is "))

---------- Buffer: Minibuffer ----------
The file is ~lewis/manual/*
---------- Buffer: Minibuffer ----------

;; Here the minibuffer is empty and only the prompt
;;   appears on its line.
(let ((insert-default-directory nil))
  (read-file-name "The file is "))

---------- Buffer: Minibuffer ----------
The file is *
---------- Buffer: Minibuffer ----------
```

`read-shell-command` *prompt* **&optional** *initial history* **&rest** *args* [Function]
This function reads a shell command from the minibuffer, prompting with *prompt* and providing intelligent completion. It completes the first word of the command using candidates that are appropriate for command names, and the rest of the command words as file names.

This function uses `minibuffer-local-shell-command-map` as the keymap for minibuffer input. The *history* argument specifies the history list to use; if is omitted or `nil`, it defaults to `shell-command-history` (see Section 21.4 [Minibuffer History], page 393). The optional argument *initial* specifies the initial content of the minibuffer (see Section 21.5 [Initial Input], page 395). The rest of *args*, if present, are used as the *default* and *inherit-input-method* arguments in `read-from-minibuffer` (see Section 21.2 [Text from Minibuffer], page 387).

`minibuffer-local-shell-command-map` [Variable]
This keymap is used by `read-shell-command` for completing command and file names that are part of a shell command. It uses `minibuffer-local-map` as its parent keymap, and binds `TAB` to `completion-at-point`.

21.6.6 Completion Variables

Here are some variables that can be used to alter the default completion behavior.

`completion-styles` [User Option]
The value of this variable is a list of completion style (symbols) to use for performing completion. A *completion style* is a set of rules for generating completions. Each symbol occurring this list must have a corresponding entry in `completion-styles-alist`.

`completion-styles-alist` [Variable]
This variable stores a list of available completion styles. Each element in the list has the form

```
(style try-completion all-completions doc)
```

Here, *style* is the name of the completion style (a symbol), which may be used in the `completion-styles` variable to refer to this style; *try-completion* is the function that does the completion; *all-completions* is the function that lists the completions; and *doc* is a string describing the completion style.

The *try-completion* and *all-completions* functions should each accept four arguments: *string*, *collection*, *predicate*, and *point*. The *string*, *collection*, and *predicate* arguments have the same meanings as in `try-completion` (see Section 21.6.1 [Basic Completion], page 396), and the *point* argument is the position of point within *string*. Each function should return a non-`nil` value if it performed its job, and `nil` if it did not (e.g., if there is no way to complete *string* according to the completion style).

When the user calls a completion command like `minibuffer-complete` (see Section 21.6.3 [Completion Commands], page 401), Emacs looks for the first style listed in `completion-styles` and calls its *try-completion* function. If this function returns `nil`, Emacs moves to the next listed completion style and calls its *try-completion* function, and so on until one of the *try-completion* functions successfully performs completion and returns a non-`nil` value. A similar procedure is used for listing completions, via the *all-completions* functions.

See Section "Completion Styles" in *The GNU Emacs Manual*, for a description of the available completion styles.

`completion-category-overrides` [User Option]
This variable specifies special completion styles and other completion behaviors to use when completing certain types of text. Its value should be an alist with elements of the form `(category . alist)`. *category* is a symbol describing what is being completed; currently, the `buffer`, `file`, and `unicode-name` categories are defined, but others can be defined via specialized completion functions (see Section 21.6.7 [Programmed Completion], page 410). *alist* is an association list describing how completion should behave for the corresponding category. The following alist keys are supported:

`styles` The value should be a list of completion styles (symbols).

`cycle` The value should be a value for `completion-cycle-threshold` (see Section "Completion Options" in *The GNU Emacs Manual*) for this category.

Additional alist entries may be defined in the future.

`completion-extra-properties` [Variable]

This variable is used to specify extra properties of the current completion command. It is intended to be let-bound by specialized completion commands. Its value should be a list of property and value pairs. The following properties are supported:

`:annotation-function`
: The value should be a function to add annotations in the completions buffer. This function must accept one argument, a completion, and should either return `nil` or a string to be displayed next to the completion. Unless this function puts own face on the annotation suffix string, the `completions-annotations` face is added by default to that string.

`:affixation-function`
: The value should be a function to add prefixes and suffixes to completions. This function must accept one argument, a list of completions, and should return a list of annotated completions. Each element of the returned list must be a three-element list, the completion, a prefix string, and a suffix string. This function takes priority over `:annotation-function`.

`:exit-function`
: The value should be a function to run after performing completion. The function should accept two arguments, *string* and *status*, where *string* is the text to which the field was completed, and *status* indicates what kind of operation happened: `finished` if text is now complete, `sole` if the text cannot be further completed but completion is not finished, or `exact` if the text is a valid completion but may be further completed.

21.6.7 Programmed Completion

Sometimes it is not possible or convenient to create an alist or an obarray containing all the intended possible completions ahead of time. In such a case, you can supply your own function to compute the completion of a given string. This is called *programmed completion*. Emacs uses programmed completion when completing file names (see Section 26.9.6 [File Name Completion], page 646), among many other cases.

To use this feature, pass a function as the *collection* argument to `completing-read`. The function `completing-read` arranges to pass your completion function along to `try-completion`, `all-completions`, and other basic completion functions, which will then let your function do all the work.

The completion function should accept three arguments:

- The string to be completed.
- A predicate function with which to filter possible matches, or `nil` if none. The function should call the predicate for each possible match, and ignore the match if the predicate returns `nil`.
- A flag specifying the type of completion operation to perform; see Section 21.6.1 [Basic Completion], page 396, for the details of those operations. This flag may be one of the following values.

  `nil` This specifies a `try-completion` operation. The function should return `nil` if there are no matches; it should return `t` if the specified string is a

unique and exact match; and it should return the longest common prefix substring of all matches otherwise.

`t` This specifies an `all-completions` operation. The function should return a list of all possible completions of the specified string.

`lambda` This specifies a `test-completion` operation. The function should return `t` if the specified string is an exact match for some completion alternative; `nil` otherwise.

`(boundaries . `*`suffix`*`)`
This specifies a `completion-boundaries` operation. The function should return `(boundaries` *`start`* `.` *`end`*`)`, where *start* is the position of the beginning boundary in the specified string, and *end* is the position of the end boundary in *suffix*.

If a Lisp program returns nontrivial boundaries, it should make sure that the `all-completions` operation is consistent with them. The completions returned by `all-completions` should only pertain to the piece of the prefix and suffix covered by the completion boundaries. See Section 21.6.1 [Basic Completion], page 396, for the precise expected semantics of completion boundaries.

`metadata` This specifies a request for information about the state of the current completion. The return value should have the form `(metadata .` *`alist`*`)`, where *alist* is an alist whose elements are described below.

If the flag has any other value, the completion function should return `nil`.

The following is a list of metadata entries that a completion function may return in response to a `metadata` flag argument:

`category` The value should be a symbol describing what kind of text the completion function is trying to complete. If the symbol matches one of the keys in `completion-category-overrides`, the usual completion behavior is overridden. See Section 21.6.6 [Completion Variables], page 408.

`annotation-function`
The value should be a function for *annotating* completions. The function should take one argument, *string*, which is a possible completion. It should return a string, which is displayed after the completion *string* in the `*Completions*` buffer. Unless this function puts own face on the annotation suffix string, the `completions-annotations` face is added by default to that string.

`affixation-function`
The value should be a function for adding prefixes and suffixes to completions. The function should take one argument, *completions*, which is a list of possible completions. It should return such a list of *completions* where each element contains a list of three elements: a completion, a prefix which is displayed before the completion string in the `*Completions*` buffer, and a suffix displayed after the completion string. This function takes priority over `annotation-function`.

`group-function`
: The value should be a function for grouping the completion candidates. The function must take two arguments, *completion*, which is a completion candidate and *transform*, which is a boolean flag. If *transform* is `nil`, the function must return the group title of the group to which the candidate belongs. The returned title can also be `nil`. Otherwise the function must return the transformed candidate. The transformation can for example remove a redundant prefix, which is displayed in the group title.

`display-sort-function`
: The value should be a function for sorting completions. The function should take one argument, a list of completion strings, and return a sorted list of completion strings. It is allowed to alter the input list destructively.

`cycle-sort-function`
: The value should be a function for sorting completions, when `completion-cycle-threshold` is non-`nil` and the user is cycling through completion alternatives. See Section "Completion Options" in *The GNU Emacs Manual*. Its argument list and return value are the same as for `display-sort-function`.

`completion-table-dynamic` *function* **&optional** *switch-buffer* [Function]
: This function is a convenient way to write a function that can act as a programmed completion function. The argument *function* should be a function that takes one argument, a string, and returns a completion table (see Section 21.6.1 [Basic Completion], page 396) containing all the possible completions. The table returned by *function* can also include elements that don't match the string argument; they are automatically filtered out by `completion-table-dynamic`. In particular, *function* can ignore its argument and return a full list of all possible completions. You can think of `completion-table-dynamic` as a transducer between *function* and the interface for programmed completion functions.

  If the optional argument *switch-buffer* is non-`nil`, and completion is performed in the minibuffer, *function* will be called with current buffer set to the buffer from which the minibuffer was entered.

  The return value of `completion-table-dynamic` is a function that can be used as the 2nd argument to `try-completion` and `all-completions`. Note that this function will always return empty metadata and trivial boundaries.

`completion-table-with-cache` *function* **&optional** *ignore-case* [Function]
: This is a wrapper for `completion-table-dynamic` that saves the last argument-result pair. This means that multiple lookups with the same argument only need to call *function* once. This can be useful when a slow operation is involved, such as calling an external process.

21.6.8 Completion in Ordinary Buffers

Although completion is usually done in the minibuffer, the completion facility can also be used on the text in ordinary Emacs buffers. In many major modes, in-buffer completion is performed by the `C-M-i` or `M-TAB` command, bound to `completion-at-point`. See Section

"Symbol Completion" in *The GNU Emacs Manual*. This command uses the abnormal hook variable `completion-at-point-functions`:

`completion-at-point-functions` [Variable]

The value of this abnormal hook should be a list of functions, which are used to compute a completion table (see Section 21.6.1 [Basic Completion], page 396) for completing the text at point. It can be used by major modes to provide mode-specific completion tables (see Section 24.2.1 [Major Mode Conventions], page 529).

When the command `completion-at-point` runs, it calls the functions in the list one by one, without any argument. Each function should return `nil` unless it can and wants to take responsibility for the completion data for the text at point. Otherwise it should return a list of the following form:

```
(start end collection . props)
```

start and *end* delimit the text to complete (which should enclose point). *collection* is a completion table for completing that text, in a form suitable for passing as the second argument to `try-completion` (see Section 21.6.1 [Basic Completion], page 396); completion alternatives will be generated from this completion table in the usual way, via the completion styles defined in `completion-styles` (see Section 21.6.6 [Completion Variables], page 408). *props* is a property list for additional information; any of the properties in `completion-extra-properties` are recognized (see Section 21.6.6 [Completion Variables], page 408), as well as the following additional ones:

`:predicate`
: The value should be a predicate that completion candidates need to satisfy.

`:exclusive`
: If the value is `no`, then if the completion table fails to match the text at point, `completion-at-point` moves on to the next function in `completion-at-point-functions` instead of reporting a completion failure.

The functions on this hook should generally return quickly, since they may be called very often (e.g., from `post-command-hook`). Supplying a function for *collection* is strongly recommended if generating the list of completions is an expensive operation. Emacs may internally call functions in `completion-at-point-functions` many times, but care about the value of *collection* for only some of these calls. By supplying a function for *collection*, Emacs can defer generating completions until necessary. You can use `completion-table-dynamic` to create a wrapper function:

```
;; Avoid this pattern.
(let ((beg ...) (end ...) (my-completions (my-make-completions)))
  (list beg end my-completions))

;; Use this instead.
(let ((beg ...) (end ...))
  (list beg
        end
        (completion-table-dynamic
          (lambda (_)
            (my-make-completions)))))
```

Additionally, the *collection* should generally not be pre-filtered based on the current text between *start* and *end*, because that is the responsibility of the caller of `completion-at-point-functions` to do that according to the completion styles it decides to use.

A function in `completion-at-point-functions` may also return a function instead of a list as described above. In that case, that returned function is called, with no argument, and it is entirely responsible for performing the completion. We discourage this usage; it is only intended to help convert old code to using `completion-at-point`.

The first function in `completion-at-point-functions` to return a non-`nil` value is used by `completion-at-point`. The remaining functions are not called. The exception to this is when there is an `:exclusive` specification, as described above.

The following function provides a convenient way to perform completion on an arbitrary stretch of text in an Emacs buffer:

`completion-in-region` *start end collection* **&optional** *predicate* [Function]

This function completes the text in the current buffer between the positions *start* and *end*, using *collection*. The argument *collection* has the same meaning as in `try-completion` (see Section 21.6.1 [Basic Completion], page 396).

This function inserts the completion text directly into the current buffer. Unlike `completing-read` (see Section 21.6.2 [Minibuffer Completion], page 399), it does not activate the minibuffer.

For this function to work, point must be somewhere between *start* and *end*.

21.7 Yes-or-No Queries

This section describes functions used to ask the user a yes-or-no question. The function `y-or-n-p` can be answered with a single character; it is useful for questions where an inadvertent wrong answer will not have serious consequences. `yes-or-no-p` is suitable for more momentous questions, since it requires three or four characters to answer.

If either of these functions is called in a command that was invoked using the mouse or some other window-system gesture, or in a command invoked via a menu, then they use a dialog box or pop-up menu to ask the question if dialog boxes are supported. Otherwise, they use keyboard input. You can force use either of the mouse or of keyboard input by binding `last-nonmenu-event` to a suitable value around the call—bind it to `t` to force keyboard interaction, and to a list to force dialog boxes.

Both `yes-or-no-p` and `y-or-n-p` use the minibuffer.

`y-or-n-p` *prompt* [Function]

This function asks the user a question, expecting input in the minibuffer. It returns `t` if the user types `y`, `nil` if the user types `n`. This function also accepts `SPC` to mean yes and `DEL` to mean no. It accepts `C-]` and `C-g` to quit, because the question uses the minibuffer and for that reason the user might try to use `C-]` to get out. The answer is a single character, with no `RET` needed to terminate it. Upper and lower case are equivalent.

"Asking the question" means printing *prompt* in the minibuffer, followed by the string '`(y or n) `'. If the input is not one of the expected answers (`y`, `n`, `SPC`, `DEL`, or

something that quits), the function responds '`Please answer y or n.`', and repeats the request.

This function actually uses the minibuffer, but does not allow editing of the answer. The cursor moves to the minibuffer while the question is being asked.

The answers and their meanings, even '`y`' and '`n`', are not hardwired, and are specified by the keymap `query-replace-map` (see Section 35.7 [Search and Replace], page 1019). In particular, if the user enters the special responses `recenter`, `scroll-up`, `scroll-down`, `scroll-other-window`, or `scroll-other-window-down` (respectively bound to *C-l*, *C-v*, *M-v*, *C-M-v* and *C-M-S-v* in `query-replace-map`), this function performs the specified window recentering or scrolling operation, and poses the question again.

If you bind `help-form` (see Section 25.6 [Help Functions], page 604) to a non-`nil` value while calling `y-or-n-p`, then pressing `help-char` causes it to evaluate `help-form` and display the result. `help-char` is automatically added to *prompt*.

`y-or-n-p-with-timeout` *prompt seconds default* [Function]

Like `y-or-n-p`, except that if the user fails to answer within *seconds* seconds, this function stops waiting and returns *default*. It works by setting up a timer; see Section 42.11 [Timers], page 1286. The argument *seconds* should be a number.

`yes-or-no-p` *prompt* [Function]

This function asks the user a question, expecting input in the minibuffer. It returns `t` if the user enters '`yes`', `nil` if the user types '`no`'. The user must type `RET` to finalize the response. Upper and lower case are equivalent.

`yes-or-no-p` starts by displaying *prompt* in the minibuffer, followed by '`(yes or no)` '. The user must type one of the expected responses; otherwise, the function responds '`Please answer yes or no.`', waits about two seconds and repeats the request.

`yes-or-no-p` requires more work from the user than `y-or-n-p` and is appropriate for more crucial decisions.

Here is an example:

```
(yes-or-no-p "Do you really want to remove everything? ")

;; After evaluation of the preceding expression,
;;    the following prompt appears,
;;    with an empty minibuffer:

---------- Buffer: minibuffer ----------
Do you really want to remove everything? (yes or no)
---------- Buffer: minibuffer ----------
```

If the user first types *y* `RET`, which is invalid because this function demands the entire word '`yes`', it responds by displaying these prompts, with a brief pause between them:

```
---------- Buffer: minibuffer ----------
Please answer yes or no.
Do you really want to remove everything? (yes or no)
---------- Buffer: minibuffer ----------
```

21.8 Asking Multiple-Choice Questions

This section describes facilities for asking the user more complex questions or several similar questions.

When you have a series of similar questions to ask, such as "Do you want to save this buffer?" for each buffer in turn, you should use `map-y-or-n-p` to ask the collection of questions, rather than asking each question individually. This gives the user certain convenient facilities such as the ability to answer the whole series at once.

`map-y-or-n-p` *prompter actor list* **&optional** *help action-alist no-cursor-in-echo-area* [Function]

This function asks the user a series of questions, reading a single-character answer in the echo area for each one.

The value of *list* specifies the objects to ask questions about. It should be either a list of objects or a generator function. If it is a function, it will be called with no arguments, and should return either the next object to ask about, or `nil`, meaning to stop asking questions.

The argument *prompter* specifies how to ask each question. If *prompter* is a string, the question text is computed like this:

```
(format prompter object)
```

where *object* is the next object to ask about (as obtained from *list*). See Section 4.7 [Formatting Strings], page 66, for more information about `format`.

If *prompter* is not a string, it should be a function of one argument (the object to ask about) and should return the question text for that object. If the value *prompter* returns is a string, that is the question to ask the user. The function can also return `t`, meaning to act on this object without asking the user, or `nil`, which means to silently ignore this object.

The argument *actor* says how to act on the objects for which the user answers yes. It should be a function of one argument, and will be called with each object from *list* for which the user answers yes.

If the argument *help* is given, it should be a list of this form:

```
(singular plural action)
```

where *singular* is a string containing a singular noun that describes a single object to be acted on, *plural* is the corresponding plural noun, and *action* is a transitive verb describing what *actor* does with the objects.

If you don't specify *help*, it defaults to the list `("object" "objects" "act on")`.

Each time a question is asked, the user can answer as follows:

y, *Y*, or `SPC`
: act on the object

n, *N*, or `DEL`
: skip the object

!
: act on all the following objects

`ESC` or *q*
: exit (skip all following objects)

`.` (period) act on the object and then exit

`C-h` get help

These are the same answers that `query-replace` accepts. The keymap `query-replace-map` defines their meaning for `map-y-or-n-p` as well as for `query-replace`; see Section 35.7 [Search and Replace], page 1019.

You can use *action-alist* to specify additional possible answers and what they mean. If provided, *action-alist* should be an alist whose elements are of the form `(`*`char function help`*`)`. Each of the alist elements defines one additional answer. In each element, *char* is a character (the answer); *function* is a function of one argument (an object from *list*); and *help* is a string. When the user responds with *char*, `map-y-or-n-p` calls *function*. If it returns non-`nil`, the object is considered to have been acted upon, and `map-y-or-n-p` advances to the next object in *list*. If it returns `nil`, the prompt is repeated for the same object. If the user requests help, the text in *help* is used to describe these additional answers.

Normally, `map-y-or-n-p` binds `cursor-in-echo-area` while prompting. But if *no-cursor-in-echo-area* is non-`nil`, it does not do that.

If `map-y-or-n-p` is called in a command that was invoked using the mouse or some other window-system gesture, or a command invoked via a menu, then it uses a dialog box or pop-up menu to ask the question if dialog boxes are supported. In this case, it does not use keyboard input or the echo area. You can force use either of the mouse or of keyboard input by binding `last-nonmenu-event` to a suitable value around the call—bind it to `t` to force keyboard interaction, and to a list to force dialog boxes.

The return value of `map-y-or-n-p` is the number of objects acted on.

If you need to ask the user a question that might have more than just 2 answers, use `read-answer`.

`read-answer` *question answers* [Function]

This function prompts the user with text in *question*, which should end in the '`SPC`' character. The function includes in the prompt the possible responses in *answers* by appending them to the end of *question*. The possible responses are provided in *answers* as an alist whose elements are of the following form:

```
(long-answer short-answer help-message)
```

where *long-answer* is the complete text of the user response, a string; *short-answer* is a short form of the same response, a single character or a function key; and *help-message* is the text that describes the meaning of the answer. If the variable `read-answer-short` is non-`nil`, the prompt will show the short variants of the possible answers and the user is expected to type the single characters/keys shown in the prompt; otherwise the prompt will show the long variants of the answers, and the user is expected to type the full text of one of the answers and end by pressing `RET`. If `use-dialog-box` is non-`nil`, and this function was invoked by mouse events, the question and the answers will be displayed in a GUI dialog box.

The function returns the text of the *long-answer* selected by the user, regardless of whether long or short answers were shown in the prompt and typed by the user.

Here is an example of using this function:

```
(let ((read-answer-short t))
  (read-answer "Foo "
     '(("yes"  ?y "perform the action")
       ("no"   ?n "skip to the next")
       ("all"  ?! "perform for the rest without more questions")
       ("help" ?h "show help")
       ("quit" ?q "exit"))))
```

`read-char-from-minibuffer` *prompt* **&optional** *chars history* [Function]
This function uses the minibuffer to read and return a single character. Optionally, it ignores any input that is not a member of *chars*, a list of accepted characters. The *history* argument specifies the history list symbol to use; if it is omitted or `nil`, this function doesn't use the history.

If you bind `help-form` (see Section 25.6 [Help Functions], page 604) to a non-`nil` value while calling `read-char-from-minibuffer`, then pressing `help-char` causes it to evaluate `help-form` and display the result.

21.9 Reading a Password

To read a password to pass to another program, you can use the function `read-passwd`.

`read-passwd` *prompt* **&optional** *confirm default* [Function]
This function reads a password, prompting with *prompt*. It does not echo the password as the user types it; instead, it echoes '`*`' for each character in the password. If you want to apply another character to hide the password, let-bind the variable `read-hide-char` with that character.

The optional argument *confirm*, if non-`nil`, says to read the password twice and insist it must be the same both times. If it isn't the same, the user has to type it over and over until the last two times match.

The optional argument *default* specifies the default password to return if the user enters empty input. If *default* is `nil`, then `read-passwd` returns the null string in that case.

21.10 Minibuffer Commands

This section describes some commands meant for use in the minibuffer.

`exit-minibuffer` [Command]
This command exits the active minibuffer. It is normally bound to keys in minibuffer local keymaps. The command throws an error if the current buffer is a minibuffer, but not the active minibuffer.

`self-insert-and-exit` [Command]
This command exits the active minibuffer after inserting the last character typed on the keyboard (found in `last-command-event`; see Section 22.5 [Command Loop Info], page 437).

`previous-history-element` *n* [Command]
This command replaces the minibuffer contents with the value of the *n*th previous (older) history element.

`next-history-element` *n* [Command]
This command replaces the minibuffer contents with the value of the *n*th more recent history element. The position in the history can go beyond the current position and invoke "future history" (see Section 21.2 [Text from Minibuffer], page 387).

`previous-matching-history-element` *pattern n* [Command]
This command replaces the minibuffer contents with the value of the *n*th previous (older) history element that matches *pattern* (a regular expression).

`next-matching-history-element` *pattern n* [Command]
This command replaces the minibuffer contents with the value of the *n*th next (newer) history element that matches *pattern* (a regular expression).

`previous-complete-history-element` *n* [Command]
This command replaces the minibuffer contents with the value of the *n*th previous (older) history element that completes the current contents of the minibuffer before the point.

`next-complete-history-element` *n* [Command]
This command replaces the minibuffer contents with the value of the *n*th next (newer) history element that completes the current contents of the minibuffer before the point.

`goto-history-element` *nabs* [Command]
This function puts element of the minibuffer history in the minibuffer. The argument *nabs* specifies the absolute history position in descending order, where 0 means the current element and a positive number *n* means the *n*th previous element. NABS being a negative number -*n* means the *n*th entry of "future history". When this function reaches the end of the default values provided by `read-from-minibuffer` (see Section 21.2 [Text from Minibuffer], page 387) and `completing-read` (see Section 21.6.2 [Minibuffer Completion], page 399), it adds the completion candidates to "future history", see Section 21.4 [minibuffer-default-add-function], page 393.

21.11 Minibuffer Windows

These functions access and select minibuffer windows, test whether they are active and control how they get resized.

`minibuffer-window` **&optional** *frame* [Function]
This function returns the minibuffer window used for frame *frame*. If *frame* is `nil`, that stands for the selected frame.

Note that the minibuffer window used by a frame need not be part of that frame—a frame that has no minibuffer of its own necessarily uses some other frame's minibuffer window. The minibuffer window of a minibuffer-less frame can be changed by setting that frame's `minibuffer` frame parameter (see Section 30.4.3.5 [Buffer Parameters], page 814).

`set-minibuffer-window` *window* [Function]
: This function specifies *window* as the minibuffer window to use. This affects where the minibuffer is displayed if you put text in it without invoking the usual minibuffer commands. It has no effect on the usual minibuffer input functions because they all start by choosing the minibuffer window according to the selected frame.

`window-minibuffer-p` **&optional** *window* [Function]
: This function returns `t` if *window* is a minibuffer window. *window* defaults to the selected window.

The following function returns the window showing the currently active minibuffer.

`active-minibuffer-window` [Function]
: This function returns the window of the currently active minibuffer, or `nil` if there is no active minibuffer.

It is not sufficient to determine whether a given window shows the currently active minibuffer by comparing it with the result of `(minibuffer-window)`, because there can be more than one minibuffer window if there is more than one frame.

`minibuffer-window-active-p` *window* [Function]
: This function returns non-`nil` if *window* shows the currently active minibuffer.

The following two options control whether minibuffer windows are resized automatically and how large they can get in the process.

`resize-mini-windows` [User Option]
: This option specifies whether minibuffer windows are resized automatically. The default value is `grow-only`, which means that a minibuffer window by default expands automatically to accommodate the text it displays and shrinks back to one line as soon as the minibuffer gets empty. If the value is `t`, Emacs will always try to fit the height of a minibuffer window to the text it displays (with a minimum of one line). If the value is `nil`, a minibuffer window never changes size automatically. In that case the window resizing commands (see Section 29.5 [Resizing Windows], page 705) can be used to adjust its height.

`max-mini-window-height` [User Option]
: This option provides a maximum height for resizing minibuffer windows automatically. A floating-point number specifies the maximum height as a fraction of the frame's height; an integer specifies the maximum height in units of the frame's canonical character height (see Section 30.3.2 [Frame Font], page 799). The default value is 0.25.

Note that the values of the above two variables take effect at display time, so let-binding them around code which produces echo-area messages will not work. If you want to prevent resizing of minibuffer windows when displaying long messages, bind the `message-truncate-lines` variable instead (see Section 41.4.4 [Echo Area Customization], page 1142).

The option `resize-mini-windows` does not affect the behavior of minibuffer-only frames (see Section 30.3.1 [Frame Layout], page 793). The following option allows to automatically resize such frames as well.

`resize-mini-frames` [User Option]

If this is `nil`, minibuffer-only frames are never resized automatically.

If this is a function, that function is called with the minibuffer-only frame to be resized as sole argument. At the time this function is called, the buffer of the minibuffer window of that frame is the buffer whose contents will be shown the next time that window is redisplayed. The function is expected to fit the frame to the buffer in some appropriate way.

Any other non-`nil` value means to resize minibuffer-only frames by calling `fit-mini-frame-to-buffer`, a function that behaves like `fit-frame-to-buffer` (see Section 29.5 [Resizing Windows], page 705) but does not strip leading or trailing empty lines from the buffer text.

21.12 Minibuffer Contents

These functions access the minibuffer prompt and contents.

`minibuffer-prompt` [Function]

This function returns the prompt string of the currently active minibuffer. If no minibuffer is active, it returns `nil`.

`minibuffer-prompt-end` [Function]

This function returns the current position of the end of the minibuffer prompt, if a minibuffer is current. Otherwise, it returns the minimum valid buffer position.

`minibuffer-prompt-width` [Function]

This function returns the current display-width of the minibuffer prompt, if a minibuffer is current. Otherwise, it returns zero.

`minibuffer-contents` [Function]

This function returns the editable contents of the minibuffer (that is, everything except the prompt) as a string, if a minibuffer is current. Otherwise, it returns the entire contents of the current buffer.

`minibuffer-contents-no-properties` [Function]

This is like `minibuffer-contents`, except that it does not copy text properties, just the characters themselves. See Section 33.19 [Text Properties], page 919.

`delete-minibuffer-contents` [Command]

This command erases the editable contents of the minibuffer (that is, everything except the prompt), if a minibuffer is current. Otherwise, it erases the entire current buffer.

21.13 Recursive Minibuffers

These functions and variables deal with recursive minibuffers (see Section 22.13 [Recursive Editing], page 476):

`minibuffer-depth` [Function]

This function returns the current depth of activations of the minibuffer, a nonnegative integer. If no minibuffers are active, it returns zero.

`enable-recursive-minibuffers` [User Option]

If this variable is non-`nil`, you can invoke commands (such as `find-file`) that use minibuffers even while the minibuffer is active. Such invocation produces a recursive editing level for a new minibuffer. By default, the outer-level minibuffer is invisible while you are editing the inner one. If you have `minibuffer-follows-selected-frame` set to `nil`, you can have minibuffers visible on several frames at the same time. See Section "Basic Minibuffer" in `emacs`.

If this variable is `nil`, you cannot invoke minibuffer commands when the minibuffer is active, not even if you switch to another window to do it.

If a command name has a property `enable-recursive-minibuffers` that is non-`nil`, then the command can use the minibuffer to read arguments even if it is invoked from the minibuffer. A command can also achieve this by binding `enable-recursive-minibuffers` to `t` in the interactive declaration (see Section 22.2.1 [Using Interactive], page 425). The minibuffer command `next-matching-history-element` (normally *`M-s`* in the minibuffer) does the latter.

21.14 Inhibiting Interaction

It's sometimes useful to be able to run Emacs as a headless server process that responds to commands given over a network connection. However, Emacs is primarily a platform for interactive usage, so many commands prompt the user for feedback in certain anomalous situations. This makes this use case more difficult, since the server process will just hang waiting for user input.

Binding the `inhibit-interaction` variable to something non-`nil` makes Emacs signal a `inhibited-interaction` error instead of prompting, which can then be used by the server process to handle these situations.

Here's a typical use case:

```
(let ((inhibit-interaction t))
  (respond-to-client
   (condition-case err
       (my-client-handling-function)
     (inhibited-interaction err))))
```

If `my-client-handling-function` ends up calling something that asks the user for something (via `y-or-n-p` or `read-from-minibuffer` or the like), an `inhibited-interaction` error is signaled instead. The server code then catches that error and reports it to the client.

21.15 Minibuffer Miscellany

`minibufferp` **&optional** *buffer-or-name live* [Function]

This function returns non-`nil` if *buffer-or-name* is a minibuffer. If *buffer-or-name* is omitted or `nil`, it tests the current buffer. When *live* is non-`nil`, the function returns non-`nil` only when *buffer-or-name* is an active minibuffer.

`minibuffer-setup-hook` [Variable]

This is a normal hook that is run whenever a minibuffer is entered. See Section 24.1 [Hooks], page 525.

`minibuffer-with-setup-hook` *function* **&rest** *body* [Macro]
This macro executes *body* after arranging for the specified *function* to be called via `minibuffer-setup-hook`. By default, *function* is called before the other functions in the `minibuffer-setup-hook` list, but if *function* is of the form `(:append` *`func`*`)`, *func* will be called *after* the other hook functions.

The *body* forms should not use the minibuffer more than once. If the minibuffer is re-entered recursively, *function* will only be called once, for the outermost use of the minibuffer.

`minibuffer-exit-hook` [Variable]
This is a normal hook that is run whenever a minibuffer is exited. See Section 24.1 [Hooks], page 525.

`minibuffer-help-form` [Variable]
The current value of this variable is used to rebind `help-form` locally inside the minibuffer (see Section 25.6 [Help Functions], page 604).

`minibuffer-scroll-window` [Variable]
If the value of this variable is non-`nil`, it should be a window object. When the function `scroll-other-window` is called in the minibuffer, it scrolls this window (see Section 29.21 [Textual Scrolling], page 765).

`minibuffer-selected-window` [Function]
This function returns the window that was selected just before the minibuffer window was selected. If the selected window is not a minibuffer window, it returns `nil`.

`minibuffer-message` *string* **&rest** *args* [Function]
This function is like `message` (see Section 41.4.1 [Displaying Messages], page 1135), but it displays the messages specially when the user types in the minibuffer, typically because Emacs prompted the user for some input. When the minibuffer is the current buffer, this function displays the message specified by *string* temporarily at the end of the minibuffer text, and thus avoids hiding the minibuffer text by the echo-area display of the message. It leaves the message on display for a few seconds, or until the next input event arrives, whichever comes first. The variable `minibuffer-message-timeout` specifies the number of seconds to wait in the absence of input. It defaults to 2. If *args* is non-`nil`, the actual message is obtained by passing *string* and *args* through `format-message`. See Section 4.7 [Formatting Strings], page 66.

If called when the minibuffer is not the current buffer, this function just calls `message`, and thus *string* will be shown in the echo-area.

`minibuffer-inactive-mode` [Command]
This is the major mode used in inactive minibuffers. It uses keymap `minibuffer-inactive-mode-map`. This can be useful if the minibuffer is in a separate frame. See Section 30.9 [Minibuffers and Frames], page 826.

22 Command Loop

When you run Emacs, it enters the *editor command loop* almost immediately. This loop reads key sequences, executes their definitions, and displays the results. In this chapter, we describe how these things are done, and the subroutines that allow Lisp programs to do them.

22.1 Command Loop Overview

The first thing the command loop must do is read a key sequence, which is a sequence of input events that translates into a command. It does this by calling the function `read-key-sequence`. Lisp programs can also call this function (see Section 22.8.1 [Key Sequence Input], page 462). They can also read input at a lower level with `read-key` or `read-event` (see Section 22.8.2 [Reading One Event], page 463), or discard pending input with `discard-input` (see Section 22.8.6 [Event Input Misc], page 469).

The key sequence is translated into a command through the currently active keymaps. See Section 23.10 [Key Lookup], page 495, for information on how this is done. The result should be a keyboard macro or an interactively callable function. If the key is `M-x`, then it reads the name of another command, which it then calls. This is done by the command `execute-extended-command` (see Section 22.3 [Interactive Call], page 433).

Prior to executing the command, Emacs runs `undo-boundary` to create an undo boundary. See Section 33.10 [Maintaining Undo], page 900.

To execute a command, Emacs first reads its arguments by calling `command-execute` (see Section 22.3 [Interactive Call], page 433). For commands written in Lisp, the `interactive` specification says how to read the arguments. This may use the prefix argument (see Section 22.12 [Prefix Command Arguments], page 474) or may read with prompting in the minibuffer (see Chapter 21 [Minibuffers], page 386). For example, the command `find-file` has an `interactive` specification which says to read a file name using the minibuffer. The function body of `find-file` does not use the minibuffer, so if you call `find-file` as a function from Lisp code, you must supply the file name string as an ordinary Lisp function argument.

If the command is a keyboard macro (i.e., a string or vector), Emacs executes it using `execute-kbd-macro` (see Section 22.16 [Keyboard Macros], page 479).

`pre-command-hook` [Variable]

This normal hook is run by the editor command loop before it executes each command. At that time, `this-command` contains the command that is about to run, and `last-command` describes the previous command. See Section 22.5 [Command Loop Info], page 437.

`post-command-hook` [Variable]

This normal hook is run by the editor command loop after it executes each command (including commands terminated prematurely by quitting or by errors). At that time, `this-command` refers to the command that just ran, and `last-command` refers to the command before that.

This hook is also run when Emacs first enters the command loop (at which point `this-command` and `last-command` are both `nil`).

Quitting is suppressed while running `pre-command-hook` and `post-command-hook`. If an error happens while executing one of these hooks, it does not terminate execution of the hook; instead the error is silenced and the function in which the error occurred is removed from the hook.

A request coming into the Emacs server (see Section "Emacs Server" in *The GNU Emacs Manual*) runs these two hooks just as a keyboard command does.

Note that, when the buffer text includes very long lines, these two hooks are called as if they were in a `with-restriction` form (see Section 31.4 [Narrowing], page 867), with a `long-line-optimizations-in-command-hooks` label and with the buffer narrowed to a portion around point.

22.2 Defining Commands

The special form `interactive` turns a Lisp function into a command. The `interactive` form must be located at top-level in the function body, usually as the first form in the body; this applies to both lambda expressions (see Section 13.2 [Lambda Expressions], page 232) and `defun` forms (see Section 13.4 [Defining Functions], page 237). This form does nothing during the actual execution of the function; its presence serves as a flag, telling the Emacs command loop that the function can be called interactively. The argument of the `interactive` form specifies how the arguments for an interactive call should be read.

Alternatively, an `interactive` form may be specified in a function symbol's `interactive-form` property. A non-`nil` value for this property takes precedence over any `interactive` form in the function body itself. This feature is seldom used.

Sometimes, a function is only intended to be called interactively, never directly from Lisp. In that case, give the function a non-`nil` `interactive-only` property, either directly or via `declare` (see Section 13.15 [Declare Form], page 263). This causes the byte compiler to warn if the command is called from Lisp. The output of `describe-function` will include similar information. The value of the property can be: a string, which the byte-compiler will use directly in its warning (it should end with a period, and not start with a capital, e.g., `"use (system-name) instead."`); `t`; any other symbol, which should be an alternative function to use in Lisp code.

Generic functions (see Section 13.8 [Generic Functions], page 245) cannot be turned into commands by adding the `interactive` form to them.

22.2.1 Using `interactive`

This section describes how to write the `interactive` form that makes a Lisp function an interactively-callable command, and how to examine a command's `interactive` form.

`interactive` **&optional** *arg-descriptor* **&rest** *modes* [Special Form]

This special form declares that a function is a command, and that it may therefore be called interactively (via `M-x` or by entering a key sequence bound to it). The argument *arg-descriptor* declares how to compute the arguments to the command when the command is called interactively.

A command may be called from Lisp programs like any other function, but then the caller supplies the arguments and *arg-descriptor* has no effect.

The `interactive` form must be located at top-level in the function body, or in the function symbol's `interactive-form` property (see Section 9.4 [Symbol Properties], page 138). It has its effect because the command loop looks for it before calling the function (see Section 22.3 [Interactive Call], page 433). Once the function is called, all its body forms are executed; at this time, if the `interactive` form occurs within the body, the form simply returns `nil` without even evaluating its argument.

The *modes* list allows specifying which modes the command is meant to be used in. See Section 22.2.4 [Command Modes], page 431, for more details about the effect of specifying *modes*, and when to use it.

By convention, you should put the `interactive` form in the function body, as the first top-level form. If there is an `interactive` form in both the `interactive-form` symbol property and the function body, the former takes precedence. The `interactive-form` symbol property can be used to add an interactive form to an existing function, or change how its arguments are processed interactively, without redefining the function.

There are three possibilities for the argument *arg-descriptor*:

- It may be omitted or `nil`; then the command is called with no arguments. This leads quickly to an error if the command requires one or more arguments.
- It may be a string; its contents are a sequence of elements separated by newlines, one for each argument[1]. Each element consists of a code character (see Section 22.2.2 [Interactive Codes], page 428) optionally followed by a prompt (which some code characters use and some ignore). Here is an example:

  ```
  (interactive "P\nbFrobnicate buffer: ")
  ```

  The code letter '`P`' sets the command's first argument to the raw command prefix (see Section 22.12 [Prefix Command Arguments], page 474). '`bFrobnicate buffer: `' prompts the user with '`Frobnicate buffer: `' to enter the name of an existing buffer, which becomes the second and final argument.

  The prompt string can use '`%`' to include previous argument values (starting with the first argument) in the prompt. This is done using `format-message` (see Section 4.7 [Formatting Strings], page 66). For example, here is how you could read the name of an existing buffer followed by a new name to give to that buffer:

  ```
  (interactive "bBuffer to rename: \nsRename buffer %s to: ")
  ```

  If '`*`' appears at the beginning of the string, then an error is signaled if the buffer is read-only.

  If '`@`' appears at the beginning of the string, and if the key sequence used to invoke the command includes any mouse events, then the window associated with the first of those events is selected before the command is run.

  If '`^`' appears at the beginning of the string, and if the command was invoked through *shift-translation*, set the mark and activate the region temporarily, or extend an already active region, before the command is run. If the command was invoked without shift-translation, and the region is temporarily active, deactivate the region before the command is run. Shift-translation is controlled on the user level by `shift-select-mode`; see Section "Shift Selection" in *The GNU Emacs Manual*.

[1] Some elements actually supply two arguments.

You can use '`*`', '`@`', and `^` together; the order does not matter. Actual reading of arguments is controlled by the rest of the prompt string (starting with the first character that is not '`*`', '`@`', or '`^`').

- It may be a Lisp expression that is not a string; then it should be a form that is evaluated to get a list of arguments to pass to the command. Usually this form will call various functions to read input from the user, most often through the minibuffer (see Chapter 21 [Minibuffers], page 386) or directly from the keyboard (see Section 22.8 [Reading Input], page 461).

  Providing point or the mark as an argument value is also common, but if you do this *and* read input (whether using the minibuffer or not), be sure to get the integer values of point or the mark after reading. The current buffer may be receiving subprocess output; if subprocess output arrives while the command is waiting for input, it could relocate point and the mark.

  Here's an example of what *not* to do:

```
(interactive
 (list (region-beginning) (region-end)
       (read-string "Foo: " nil 'my-history)))
```

  Here's how to avoid the problem, by examining point and the mark after reading the keyboard input:

```
(interactive
 (let ((string (read-string "Foo: " nil 'my-history)))
   (list (region-beginning) (region-end) string)))
```

  Warning: the argument values should not include any data types that can't be printed and then read. Some facilities save `command-history` in a file to be read in the subsequent sessions; if a command's arguments contain a data type that prints using '`#<...>`' syntax, those facilities won't work.

  There are, however, a few exceptions: it is ok to use a limited set of expressions such as `(point)`, `(mark)`, `(region-beginning)`, and `(region-end)`, because Emacs recognizes them specially and puts the expression (rather than its value) into the command history. To see whether the expression you wrote is one of these exceptions, run the command, then examine `(car command-history)`.

`interactive-form` *function* [Function]

This function returns the `interactive` form of *function*. If *function* is an interactively callable function (see Section 22.3 [Interactive Call], page 433), the value is the command's `interactive` form `(interactive` *spec*`)`, which specifies how to compute its arguments. Otherwise, the value is `nil`. If *function* is a symbol, its function definition is used. When called on an OClosure, the work is delegated to the generic function `oclosure-interactive-form`.

`oclosure-interactive-form` *function* [Function]

Just like `interactive-form`, this function takes a command and returns its interactive form. The difference is that it is a generic function and it is only called when *function* is an OClosure (see Section 13.11 [OClosures], page 250). The purpose is to make it possible for some OClosure types to compute their interactive forms dynamically instead of carrying it in one of their slots.

This is used for example for `kmacro` functions in order to reduce their memory size, since they all share the same interactive form. It is also used for `advice` functions, where the interactive form is computed from the interactive forms of its components, so as to make this computation more lazily and to correctly adjust the interactive form when one of its component's is redefined.

22.2.2 Code Characters for `interactive`

The code character descriptions below contain a number of key words, defined here as follows:

Completion
: Provide completion. `TAB`, `SPC`, and `RET` perform name completion because the argument is read using `completing-read` (see Section 21.6 [Completion], page 396). `?` displays a list of possible completions.

Existing
: Require the name of an existing object. An invalid name is not accepted; the commands to exit the minibuffer do not exit if the current input is not valid.

Default
: A default value of some sort is used if the user enters no text in the minibuffer. The default depends on the code character.

No I/O
: This code letter computes an argument without reading any input. Therefore, it does not use a prompt string, and any prompt string you supply is ignored.

  Even though the code letter doesn't use a prompt string, you must follow it with a newline if it is not the last code character in the string.

Prompt
: A prompt immediately follows the code character. The prompt ends either with the end of the string or with a newline.

Special
: This code character is meaningful only at the beginning of the interactive string, and it does not look for a prompt or a newline. It is a single, isolated character.

Here are the code character descriptions for use with `interactive`:

'`*`'
: Signal an error if the current buffer is read-only. Special.

'`@`'
: Select the window mentioned in the first mouse event in the key sequence that invoked this command. Special.

'`^`'
: If the command was invoked through shift-translation, set the mark and activate the region temporarily, or extend an already active region, before the command is run. If the command was invoked without shift-translation, and the region is temporarily active, deactivate the region before the command is run. Special.

'`a`'
: A function name (i.e., a symbol satisfying `fboundp`). Existing, Completion, Prompt.

'`b`'
: The name of an existing buffer. By default, uses the name of the current buffer (see Chapter 28 [Buffers], page 673). Existing, Completion, Default, Prompt.

'`B`'
: A buffer name. The buffer need not exist. By default, uses the name of a recently used buffer other than the current buffer. Completion, Default, Prompt.

'`c`'
: A character. The cursor does not move into the echo area. Prompt.

‘C’ A command name (i.e., a symbol satisfying `commandp`). Existing, Completion, Prompt.

‘d’ The position of point, as an integer (see Section 31.1 [Point], page 855). No I/O.

‘D’ A directory. The default is the current default directory of the current buffer, `default-directory` (see Section 26.9.4 [File Name Expansion], page 641). Existing, Completion, Default, Prompt.

‘e’ The first or next non-keyboard event in the key sequence that invoked the command. More precisely, ‘e’ gets events that are lists, so you can look at the data in the lists. See Section 22.7 [Input Events], page 441. No I/O.

You use ‘e’ for mouse events and for special system events (see Section 22.7.12 [Misc Events], page 451). The event list that the command receives depends on the event. See Section 22.7 [Input Events], page 441, which describes the forms of the list for each event in the corresponding subsections.

You can use ‘e’ more than once in a single command's interactive specification. If the key sequence that invoked the command has *n* events that are lists, the *n*th ‘e’ provides the *n*th such event. Events that are not lists, such as function keys and ASCII characters, do not count where ‘e’ is concerned.

‘f’ A file name of an existing file (see Section 26.9 [File Names], page 636). See Section 21.6.5 [Reading File Names], page 405, for details about default values. Existing, Completion, Default, Prompt.

‘F’ A file name. The file need not exist. Completion, Default, Prompt.

‘G’ A file name. The file need not exist. If the user enters just a directory name, then the value is just that directory name, with no file name within the directory added. Completion, Default, Prompt.

‘i’ An irrelevant argument. This code always supplies `nil` as the argument's value. No I/O.

‘k’ A key sequence (see Section 23.1 [Key Sequences], page 481). This keeps reading events until a command (or undefined command) is found in the current key maps. The key sequence argument is represented as a string or vector. The cursor does not move into the echo area. Prompt.

If ‘k’ reads a key sequence that ends with a down-event, it also reads and discards the following up-event. You can get access to that up-event with the ‘U’ code character.

This kind of input is used by commands such as `describe-key` and `keymap-global-set`.

‘K’ A key sequence on a form that can be used as input to functions like `keymap-set`. This works like ‘k’, except that it suppresses, for the last input event in the key sequence, the conversions that are normally used (when necessary) to convert an undefined key into a defined one (see Section 22.8.1 [Key Sequence Input], page 462), so this form is usually used when prompting for a new key sequence that is to be bound to a command.

‘m’ The position of the mark, as an integer. No I/O.

‘M’ Arbitrary text, read in the minibuffer using the current buffer’s input method, and returned as a string (see Section “Input Methods” in *The GNU Emacs Manual*). Prompt.

‘n’ A number, read with the minibuffer. If the input is not a number, the user has to try again. ‘n’ never uses the prefix argument. Prompt.

‘N’ The numeric prefix argument; but if there is no prefix argument, read a number as with *n*. The value is always a number. See Section 22.12 [Prefix Command Arguments], page 474. Prompt.

‘p’ The numeric prefix argument. (Note that this ‘p’ is lower case.) No I/O.

‘P’ The raw prefix argument. (Note that this ‘P’ is upper case.) No I/O.

‘r’ Point and the mark, as two numeric arguments, smallest first. This is the only code letter that specifies two successive arguments rather than one. This will signal an error if the mark is not set in the buffer which is current when the command is invoked. If Transient Mark mode is turned on (see Section 32.7 [The Mark], page 875) — as it is by default — and user option `mark-even-if-inactive` is `nil`, Emacs will signal an error even if the mark *is* set, but is inactive. No I/O.

‘s’ Arbitrary text, read in the minibuffer and returned as a string (see Section 21.2 [Text from Minibuffer], page 387). Terminate the input with either *C-j* or `RET`. (*C-q* may be used to include either of these characters in the input.) Prompt.

‘S’ An interned symbol whose name is read in the minibuffer. Terminate the input with either *C-j* or `RET`. Other characters that normally terminate a symbol (e.g., whitespace, parentheses and brackets) do not do so here. Prompt.

‘U’ A key sequence or `nil`. Can be used after a ‘k’ or ‘K’ argument to get the up-event that was discarded (if any) after ‘k’ or ‘K’ read a down-event. If no up-event has been discarded, ‘U’ provides `nil` as the argument. No I/O.

‘v’ A variable declared to be a user option (i.e., satisfying the predicate `custom-variable-p`). This reads the variable using `read-variable`. See [Definition of read-variable], page 405. Existing, Completion, Prompt.

‘x’ A Lisp object, specified with its read syntax, terminated with a *C-j* or `RET`. The object is not evaluated. See Section 21.3 [Object from Minibuffer], page 392. Prompt.

‘X’ A Lisp form’s value. ‘X’ reads as ‘x’ does, then evaluates the form so that its value becomes the argument for the command. Prompt.

‘z’ A coding system name (a symbol). If the user enters null input, the argument value is `nil`. See Section 34.10 [Coding Systems], page 980. Completion, Existing, Prompt.

‘Z’ A coding system name (a symbol)—but only if this command has a prefix argument. With no prefix argument, ‘Z’ provides `nil` as the argument value. Completion, Existing, Prompt.

22.2.3 Examples of Using `interactive`

Here are some examples of `interactive`:

```
(defun foo1 ()              ; foo1 takes no arguments,
    (interactive)           ;    just moves forward two words.
    (forward-word 2))
     ⇒ foo1

(defun foo2 (n)             ; foo2 takes one argument,
    (interactive "^p")      ;    which is the numeric prefix.
                            ; under shift-select-mode,
                            ;    will activate or extend region.
    (forward-word (* 2 n)))
     ⇒ foo2

(defun foo3 (n)             ; foo3 takes one argument,
    (interactive "nCount:") ;    which is read with the Minibuffer.
    (forward-word (* 2 n)))
     ⇒ foo3

(defun three-b (b1 b2 b3)
  "Select three existing buffers.
Put them into three windows, selecting the last one."
    (interactive "bBuffer1:\nbBuffer2:\nbBuffer3:")
    (delete-other-windows)
    (split-window (selected-window) 8)
    (switch-to-buffer b1)
    (other-window 1)
    (split-window (selected-window) 8)
    (switch-to-buffer b2)
    (other-window 1)
    (switch-to-buffer b3))
     ⇒ three-b
(three-b "*scratch*" "declarations.texi" "*mail*")
     ⇒ nil
```

22.2.4 Specifying Modes For Commands

Many commands in Emacs are general, and not tied to any specific mode. For instance, *`M-x kill-region`* can be used in pretty much any mode that has editable text, and commands that display information (like *`M-x list-buffers`*) can be used in pretty much any context.

Many other commands, however, are specifically tied to a mode, and make no sense outside of that context. For instance, `M-x dired-diff` will just signal an error if used outside of a Dired buffer.

Emacs therefore has a mechanism for specifying what mode (or modes) a command "belongs" to:

```
(defun dired-diff (...)
```

```
...
(interactive "p" dired-mode)
...)
```

This will mark the command as applicable to `dired-mode` only (or any modes that are derived from `dired-mode`). Any number of modes can be added to the `interactive` form.

Specifying modes affects command completion in *M-S-x* (`execute-extended-command-for-buffer`, see Section 22.3 [Interactive Call], page 433). It may also affect completion in *M-x*, depending on the value of `read-extended-command-predicate`.

For instance, when using the `command-completion-default-include-p` predicate as the value of `read-extended-command-predicate`, *M-x* won't list commands that have been marked as being applicable to a specific mode (unless you are in a buffer that uses that mode, of course). This goes for both major and minor modes. (By contrast, *M-S-x* always omits inapplicable commands from the completion candidates.)

By default, `read-extended-command-predicate` is `nil`, and completion in *M-x* lists all the commands that match what the user has typed, whether those commands are or aren't marked as applicable to the current buffer's mode.

Marking commands to be applicable to a mode will also make *C-h m* list these commands (if they aren't bound to any keys).

If using this extended `interactive` form isn't convenient (because the code is supposed to work in older versions of Emacs that don't support the extended `interactive` form), the following equivalent declaration (see Section 13.15 [Declare Form], page 263) can be used instead:

```
(declare (modes dired-mode))
```

Which commands to tag with modes is to some degree a matter of taste, but commands that clearly do not work outside of the mode should be tagged. This includes commands that will signal an error if called from somewhere else, but also commands that are destructive when called from an unexpected mode. (This usually includes most of the commands that are written for special (i.e., non-editing) modes.)

Some commands may be harmless, and "work" when called from other modes, but should still be tagged with a mode if they don't actually make much sense to use elsewhere. For instance, many special modes have commands to exit the buffer bound to `q`, and may not do anything but issue a message like `"Goodbye from this mode"` and then call `kill-buffer`. This command will "work" from any mode, but it is highly unlikely that anybody would actually want to use the command outside the context of this special mode.

Many modes have a set of different commands that start the mode in different ways (e.g., `eww-open-in-new-buffer` and `eww-open-file`). Commands like that should never be tagged as mode-specific, as they can be issued by the user from pretty much any context.

22.2.5 Select among Command Alternatives

Sometimes it is useful to define a command that serves as a "generic dispatcher" capable of invoking one of a set of commands according to the user's needs. For example, imagine that you want to define a command named '`open`' that can "open" and display several different types of objects. Or you could have a command named '`mua`' (which stands for Mail User Agent) that can read and send email using one of several email backends, such

as Rmail, Gnus, or MH-E. The macro `define-alternatives` can be used to define such *generic commands*. A generic command is an interactive function whose implementation can be selected from several alternatives, as a matter of user preference.

`define-alternatives` *command* **&rest** *customizations* [Macro]

This macro defines the new generic *command*, which can have several alternative implementations. The argument *command* should be an unquoted symbol.

When invoked, the macro creates an interactive Lisp closure (see Section 13.10 [Closures], page 250). When the user runs *`M-x command`* `RET` for the first time, Emacs asks to select one of the alternative implementations of *command*, offering completion for the names of these alternatives. These names come from the user option whose name is *`command`*`-alternatives`, which the macro creates (if it didn't exist before). To be useful, this variable's value should be an alist whose elements have the form (*`alt-name . alt-func`*), where *alt-name* is the name of the alternative and *alt-func* is the interactive function to be called if this alternative is selected. When the user selects an alternative, Emacs remembers the selection, and will thereafter automatically call that selected alternative without prompting when the user invokes *`M-x command`* again. To choose a different alternative, type *`C-u M-x command`* `RET`–then Emacs will again prompt for one of the alternatives, and the selection will override the previous one.

The variable *`command`*`-alternatives` can be created before calling `define-alternatives`, with the appropriate values; otherwise the macro creates the variable with a `nil` value, and it should then be populated with the associations describing the alternatives. Packages that wish to provide their own implementation of an existing generic command can use `autoload` cookies (see Section 16.5 [Autoload], page 305) to add to the alist, for example:

```
;;;###autoload (push '("My name" . my-foo-symbol) foo-alternatives
```

If the optional argument *customizations* is non-`nil`, it should consist of alternating `defcustom` keywords (typically `:group` and `:version`) and values to add to the definition of the `defcustom` *`command`*`-alternatives`.

Here is an example of a simple generic dispatcher command named `open` with 3 alternative implementations:

```
(define-alternatives open
  :group 'files
  :version "42.1")
(setq open-alternatives
      '(("file" . find-file)
("directory" . dired)
("hexl" . hexl-find-file)))
```

22.3 Interactive Call

After the command loop has translated a key sequence into a command, it invokes that command using the function `command-execute`. If the command is a function, `command-execute` calls `call-interactively`, which reads the arguments and calls the command. You can also call these functions yourself.

Note that the term "command", in this context, refers to an interactively callable function (or function-like object), or a keyboard macro. It does not refer to the key sequence used to invoke a command (see Chapter 23 [Keymaps], page 481).

`commandp` *object* **&optional** *for-call-interactively* [Function]

This function returns `t` if *object* is a command. Otherwise, it returns `nil`.

Commands include strings and vectors (which are treated as keyboard macros), lambda expressions that contain a top-level `interactive` form (see Section 22.2.1 [Using Interactive], page 425), byte-code function objects made from such lambda expressions, autoload objects that are declared as interactive (non-`nil` fourth argument to `autoload`), and some primitive functions. Also, a symbol is considered a command if it has a non-`nil` `interactive-form` property, or if its function definition satisfies `commandp`.

If *for-call-interactively* is non-`nil`, then `commandp` returns `t` only for objects that `call-interactively` could call—thus, not for keyboard macros.

See `documentation` in Section 25.2 [Accessing Documentation], page 598, for a realistic example of using `commandp`.

`call-interactively` *command* **&optional** *record-flag keys* [Function]

This function calls the interactively callable function *command*, providing arguments according to its interactive calling specifications. It returns whatever *command* returns.

If, for instance, you have a function with the following signature:

```
(defun foo (begin end)
  (interactive "r")
  ...)
```

then saying

```
(call-interactively 'foo)
```

will call `foo` with the region (`point` and `mark`) as the arguments.

An error is signaled if *command* is not a function or if it cannot be called interactively (i.e., is not a command). Note that keyboard macros (strings and vectors) are not accepted, even though they are considered commands, because they are not functions. If *command* is a symbol, then `call-interactively` uses its function definition.

If *record-flag* is non-`nil`, then this command and its arguments are unconditionally added to the list `command-history`. Otherwise, the command is added only if it uses the minibuffer to read an argument. See Section 22.15 [Command History], page 478.

The argument *keys*, if given, should be a vector which specifies the sequence of events to supply if the command inquires which events were used to invoke it. If *keys* is omitted or `nil`, the default is the return value of `this-command-keys-vector`. See [Definition of this-command-keys-vector], page 438.

`funcall-interactively` *function* **&rest** *arguments* [Function]

This function works like `funcall` (see Section 13.5 [Calling Functions], page 239), but it makes the call look like an interactive invocation: a call to `called-interactively-p` inside *function* will return `t`. If *function* is not a command, it is called without signaling an error.

`command-execute` *command* **&optional** *record-flag keys special* [Function]
This function executes *command*. The argument *command* must satisfy the `commandp` predicate; i.e., it must be an interactively callable function or a keyboard macro.

A string or vector as *command* is executed with `execute-kbd-macro`. A function is passed to `call-interactively` (see above), along with the *record-flag* and *keys* arguments.

If *command* is a symbol, its function definition is used in its place. A symbol with an `autoload` definition counts as a command if it was declared to stand for an interactively callable function. Such a definition is handled by loading the specified library and then rechecking the definition of the symbol.

The argument *special*, if given, means to ignore the prefix argument and not clear it. This is used for executing special events (see Section 22.9 [Special Events], page 471).

`execute-extended-command` *prefix-argument* [Command]
This function reads a command name from the minibuffer using `completing-read` (see Section 21.6 [Completion], page 396). Then it uses `command-execute` to call the specified command. Whatever that command returns becomes the value of `execute-extended-command`.

If the command asks for a prefix argument, it receives the value *prefix-argument*. If `execute-extended-command` is called interactively, the current raw prefix argument is used for *prefix-argument*, and thus passed on to whatever command is run.

`execute-extended-command` is the normal definition of `M-x`, so it uses the string '`M-x` ' as a prompt. (It would be better to take the prompt from the events used to invoke `execute-extended-command`, but that is painful to implement.) A description of the value of the prefix argument, if any, also becomes part of the prompt.

```
(execute-extended-command 3)
---------- Buffer: Minibuffer ----------
3 M-x forward-word RET
---------- Buffer: Minibuffer ----------
     ⇒ t
```

This command heeds the `read-extended-command-predicate` variable, which can filter out commands that are not applicable to the current major mode (or enabled minor modes). By default, the value of this variable is `nil`, and no commands are filtered out. However, customizing it to invoke the function `command-completion-default-include-p` will perform mode-dependent filtering. `read-extended-command-predicate` can be any predicate function; it will be called with two parameters: the command's symbol and the current buffer. If should return non-`nil` if the command is to be included when completing in that buffer.

`execute-extended-command-for-buffer` *prefix-argument* [Command]
This is like `execute-extended-command`, but limits the commands offered for completion to those commands that are of particular relevance to the current major mode (and enabled minor modes). This includes commands that are tagged with the modes (see Section 22.2.1 [Using Interactive], page 425), and also commands that are bound to locally active keymaps. This command is the normal definition of `M-S-x` (that's "meta shift x").

Both these commands prompt for a command name, but with different completion rules. You can toggle between these two modes by using the *M-S-x* command while being prompted.

22.4 Distinguish Interactive Calls

Sometimes a command should display additional visual feedback (such as an informative message in the echo area) for interactive calls only. There are three ways to do this. The recommended way to test whether the function was called using `call-interactively` is to give it an optional argument `print-message` and use the `interactive` spec to make it non-`nil` in interactive calls. Here's an example:

```
(defun foo (&optional print-message)
  (interactive "p")
  (when print-message
    (message "foo")))
```

We use `"p"` because the numeric prefix argument is never `nil`. Defined in this way, the function does display the message when called from a keyboard macro.

The above method with the additional argument is usually best, because it allows callers to say "treat this call as interactive". But you can also do the job by testing `called-interactively-p`.

`called-interactively-p` *kind* [Function]

This function returns `t` when the calling function was called using `call-interactively`.

The argument *kind* should be either the symbol `interactive` or the symbol `any`. If it is `interactive`, then `called-interactively-p` returns `t` only if the call was made directly by the user—e.g., if the user typed a key sequence bound to the calling function, but *not* if the user ran a keyboard macro that called the function (see Section 22.16 [Keyboard Macros], page 479). If *kind* is `any`, `called-interactively-p` returns `t` for any kind of interactive call, including keyboard macros.

If in doubt, use `any`; the only known proper use of `interactive` is if you need to decide whether to display a helpful message while a function is running.

A function is never considered to be called interactively if it was called via Lisp evaluation (or with `apply` or `funcall`).

Here is an example of using `called-interactively-p`:

```
(defun foo ()
  (interactive)
  (when (called-interactively-p 'any)
    (message "Interactive!")
    'foo-called-interactively))

;; Type M-x foo.
     ⊣ Interactive!

(foo)
     ⇒ nil
```

Here is another example that contrasts direct and indirect calls to `called-interactively-p`.

```
(defun bar ()
  (interactive)
  (message "%s" (list (foo) (called-interactively-p 'any))))

;; Type M-x bar.
     ⊣ (nil t)
```

22.5 Information from the Command Loop

The editor command loop sets several Lisp variables to keep status records for itself and for commands that are run. With the exception of `this-command` and `last-command` it's generally a bad idea to change any of these variables in a Lisp program.

`last-command` [Variable]

This variable records the name of the previous command executed by the command loop (the one before the current command). Normally the value is a symbol with a function definition, but this is not guaranteed.

The value is copied from `this-command` when a command returns to the command loop, except when the command has specified a prefix argument for the following command.

This variable is always local to the current terminal and cannot be buffer-local. See Section 30.2 [Multiple Terminals], page 789.

`real-last-command` [Variable]

This variable is set up by Emacs just like `last-command`, but never altered by Lisp programs.

`last-repeatable-command` [Variable]

This variable stores the most recently executed command that was not part of an input event. This is the command `repeat` will try to repeat, See Section "Repeating" in *The GNU Emacs Manual*.

`this-command` [Variable]

This variable records the name of the command now being executed by the editor command loop. Like `last-command`, it is normally a symbol with a function definition.

The command loop sets this variable just before running a command, and copies its value into `last-command` when the command finishes (unless the command specified a prefix argument for the following command).

Some commands set this variable during their execution, as a flag for whatever command runs next. In particular, the functions for killing text set `this-command` to `kill-region` so that any kill commands immediately following will know to append the killed text to the previous kill.

If you do not want a particular command to be recognized as the previous command in the case where it got an error, you must code that command to prevent this. One way is

to set `this-command` to `t` at the beginning of the command, and set `this-command` back to its proper value at the end, like this:

```
(defun foo (args...)
  (interactive ...)
  (let ((old-this-command this-command))
    (setq this-command t)
    ...do the work...
    (setq this-command old-this-command)))
```

We do not bind `this-command` with `let` because that would restore the old value in case of error—a feature of `let` which in this case does precisely what we want to avoid.

`this-original-command` [Variable]

This has the same value as `this-command` except when command remapping occurs (see Section 23.14 [Remapping Commands], page 504). In that case, `this-command` gives the command actually run (the result of remapping), and `this-original-command` gives the command that was specified to run but remapped into another command.

`current-minibuffer-command` [Variable]

This has the same value as `this-command`, but is bound recursively when entering a minibuffer. This variable can be used from minibuffer hooks and the like to determine what command opened the current minibuffer session.

`this-command-keys` [Function]

This function returns a string or vector containing the key sequence that invoked the present command. Any events read by the command using `read-event` without a timeout get tacked on to the end.

However, if the command has called `read-key-sequence`, it returns the last read key sequence. See Section 22.8.1 [Key Sequence Input], page 462. The value is a string if all events in the sequence were characters that fit in a string. See Section 22.7 [Input Events], page 441.

```
(this-command-keys)
;; Now use C-u C-x C-e to evaluate that.
     ⇒ "^X^E"
```

`this-command-keys-vector` [Function]

Like `this-command-keys`, except that it always returns the events in a vector, so you don't need to deal with the complexities of storing input events in a string (see Section 22.7.17 [Strings of Events], page 460).

`clear-this-command-keys` **&optional** *keep-record* [Function]

This function empties out the table of events for `this-command-keys` to return. Unless *keep-record* is non-`nil`, it also empties the records that the function `recent-keys` (see Section 42.13.2 [Recording Input], page 1291) will subsequently return. This is useful after reading a password, to prevent the password from echoing inadvertently as part of the next command in certain cases.

`last-nonmenu-event` [Variable]
: This variable holds the last input event read as part of a key sequence, not counting events resulting from mouse menus.

  One use of this variable is for telling `x-popup-menu` where to pop up a menu. It is also used internally by `y-or-n-p` (see Section 21.7 [Yes-or-No Queries], page 414).

`last-command-event` [Variable]
: This variable is set to the last input event that was read by the command loop as part of a command. The principal use of this variable is in `self-insert-command`, which uses it to decide which character to insert, and in `post-self-insert-hook` (see Section 33.5 [Commands for Insertion], page 886), which uses it to access the character that was just inserted.

  ```
  last-command-event
  ;; Now use C-u C-x C-e to evaluate that.
       ⇒ 5
  ```

  The value is 5 because that is the ASCII code for `C-e`.

`last-event-frame` [Variable]
: This variable records which frame the last input event was directed to. Usually this is the frame that was selected when the event was generated, but if that frame has redirected input focus to another frame, the value is the frame to which the event was redirected. See Section 30.10 [Input Focus], page 827.

  If the last event came from a keyboard macro, the value is `macro`.

Input events must come from somewhere; sometimes, that is a keyboard macro, a signal, or 'unread-command-events', but it is usually a physical input device connected to a computer that is controlled by the user. Those devices are referred to as *input devices*, and Emacs associates each input event with the input device from which it originated. They are identified by a name that is unique to each input device.

The ability to determine the precise input device used depends on the details of each system. When that information is unavailable, Emacs reports keyboard events as originating from the '`"Virtual core keyboard"`', and other events as originating from the '`"Virtual core pointer"`'. (These values are used on every platform because the X server reports them when detailed device information is not known.)

`last-event-device` [Variable]
: This variable records the name of the input device from which the last input event read was generated. It is `nil` if no such device exists, i.e., the last input event was read from `unread-command-events`, or it came from a keyboard macro.

  When the X Input Extension is being used on X Windows, the device name is a string that is unique to each physical keyboard, pointing device and touchscreen attached to the X server. Otherwise, it is either the string '`"Virtual core pointer"`' or '`"Virtual core keyboard"`', depending on whether the event was generated by a pointing device (such as a mouse) or a keyboard.

`device-class` *frame name* [Function]

There are various different types of devices, which can be determined from their names. This function can be used to determined the correct type of the device *name* for an event originating from *frame*.

The return value is one of the following symbols ("device classes"):

`core-keyboard`
: The core keyboard; this is means the device is a keyboard-like device, but no other characteristics are unknown.

`core-pointer`
: The core pointer; this means the device is a pointing device, but no other characteristics are known.

`mouse` A computer mouse.

`trackpoint`
: A trackpoint or joystick (or other similar control.)

`eraser` The other end of a stylus on a graphics tablet, or a standalone eraser.

`pen` The pointed end of a pen on a graphics tablet, a stylus, or some other similar device.

`puck` A device that looks like a computer mouse, but reports absolute coordinates relative to some other surface.

`power-button`
: A power button or volume button (or other similar control.)

`keyboard` A computer keyboard.

`touchscreen`
: A computer touchpad.

`pad` A collection of sensitive buttons, rings, and strips commonly found around a drawing tablet.

`touchpad` An indirect touch device such as a touchpad.

`piano` A musical instrument such as an electronic keyboard.

`test` A device used by the XTEST extension to report input.

22.6 Adjusting Point After Commands

Emacs cannot display the cursor when point is in the middle of a sequence of text that has the `display` or `composition` property, or is invisible. Therefore, after a command finishes and returns to the command loop, if point is within such a sequence, the command loop normally moves point to the edge of the sequence, making this sequence effectively intangible.

A command can inhibit this feature by setting the variable `disable-point-adjustment`:

`disable-point-adjustment` [Variable]

If this variable is non-`nil` when a command returns to the command loop, then the command loop does not check for those text properties, and does not move point out of sequences that have them.

The command loop sets this variable to `nil` before each command, so if a command sets it, the effect applies only to that command.

`global-disable-point-adjustment` [Variable]

If you set this variable to a non-`nil` value, the feature of moving point out of these sequences is completely turned off.

22.7 Input Events

The Emacs command loop reads a sequence of *input events* that represent keyboard or mouse activity, or system events sent to Emacs. The events for keyboard activity are characters or symbols; other events are always lists. This section describes the representation and meaning of input events in detail.

`eventp` *object* [Function]

This function returns non-`nil` if *object* is an input event or event type.

Note that any non-`nil` symbol might be used as an event or an event type; `eventp` cannot distinguish whether a symbol is intended by Lisp code to be used as an event.

22.7.1 Keyboard Events

There are two kinds of input you can get from the keyboard: ordinary keys, and function keys. Ordinary keys correspond to (possibly modified) characters; the events they generate are represented in Lisp as characters. The event type of a *character event* is the character itself (an integer), which might have some modifier bits set; see Section 22.7.14 [Classifying Events], page 456.

An input character event consists of a *basic code* between 0 and 524287, plus any or all of these *modifier bits*:

meta The 2^{27} bit in the character code indicates a character typed with the meta key held down.

control The 2^{26} bit in the character code indicates a non-ASCII control character.

ASCII control characters such as `C-a` have special basic codes of their own, so Emacs needs no special bit to indicate them. Thus, the code for `C-a` is just 1.

But if you type a control combination not in ASCII, such as `%` with the control key, the numeric value you get is the code for `%` plus 2^{26} (assuming the terminal supports non-ASCII control characters), i.e. with the 27th bit set.

shift The 2^{25} bit (the 26th bit) in the character event code indicates an ASCII control character typed with the shift key held down.

For letters, the basic code itself indicates upper versus lower case; for digits and punctuation, the shift key selects an entirely different character with a different basic code. In order to keep within the ASCII character set whenever possible, Emacs avoids using the 2^{25} bit for those character events.

However, ASCII provides no way to distinguish `C-A` from `C-a`, so Emacs uses the 2^{25} bit in `C-A` and not in `C-a`.

hyper
: The 2^{24} bit in the character event code indicates a character typed with the hyper key held down.

super
: The 2^{23} bit in the character event code indicates a character typed with the super key held down.

alt
: The 2^{22} bit in the character event code indicates a character typed with the alt key held down. (The key labeled `Alt` on most keyboards is actually treated as the meta key, not this.)

It is best to avoid mentioning specific bit numbers in your program. To test the modifier bits of a character, use the function `event-modifiers` (see Section 22.7.14 [Classifying Events], page 456). When making key bindings with `keymap-set`, you specify these events using strings like '`C-H-x`' instead (for "control hyper x") (see Section 23.12 [Changing Key Bindings], page 498).

22.7.2 Function Keys

Most keyboards also have *function keys*—keys that have names or symbols that are not characters. Function keys are represented in Emacs Lisp as symbols; the symbol's name is the function key's label, in lower case. For example, pressing a key labeled `F1` generates an input event represented by the symbol `f1`.

The event type of a function key event is the event symbol itself. See Section 22.7.14 [Classifying Events], page 456.

Here are a few special cases in the symbol-naming convention for function keys:

`backspace`, `tab`, `newline`, `return`, `delete`
: These keys correspond to common ASCII control characters that have special keys on most keyboards.

  In ASCII, `C-i` and `TAB` are the same character. If the terminal can distinguish between them, Emacs conveys the distinction to Lisp programs by representing the former as the integer 9, and the latter as the symbol `tab`.

  Most of the time, it's not useful to distinguish the two. So normally `local-function-key-map` (see Section 23.15 [Translation Keymaps], page 505) is set up to map `tab` into 9. Thus, a key binding for character code 9 (the character `C-i`) also applies to `tab`. Likewise for the other symbols in this group. The function `read-char` likewise converts these events into characters.

  In ASCII, `BS` is really `C-h`. But `backspace` converts into the character code 127 (`DEL`), not into code 8 (`BS`). This is what most users prefer.

`left`, `up`, `right`, `down`
: Cursor arrow keys

`kp-add`, `kp-decimal`, `kp-divide`, ...
: Keypad keys (to the right of the regular keyboard).

`kp-0`, `kp-1`, ...
: Keypad keys with digits.

`kp-f1`, `kp-f2`, `kp-f3`, `kp-f4`
: Keypad PF keys.

`kp-home`, `kp-left`, `kp-up`, `kp-right`, `kp-down`
: Keypad arrow keys. Emacs normally translates these into the corresponding non-keypad keys `home`, `left`, ...

`kp-prior`, `kp-next`, `kp-end`, `kp-begin`, `kp-insert`, `kp-delete`
: Additional keypad duplicates of keys ordinarily found elsewhere. Emacs normally translates these into the like-named non-keypad keys.

You can use the modifier keys `ALT`, `CTRL`, `HYPER`, `META`, `SHIFT`, and `SUPER` with function keys. The way to represent them is with prefixes in the symbol name:

'`A-`' The alt modifier.

'`C-`' The control modifier.

'`H-`' The hyper modifier.

'`M-`' The meta modifier.

'`S-`' The shift modifier.

'`s-`' The super modifier.

Thus, the symbol for the key `F3` with `META` held down is `M-f3`. When you use more than one prefix, we recommend you write them in alphabetical order; but the order does not matter in arguments to the key-binding lookup and modification functions.

22.7.3 Mouse Events

Emacs supports four kinds of mouse events: click events, drag events, button-down events, and motion events. All mouse events are represented as lists. The CAR of the list is the event type; this says which mouse button was involved, and which modifier keys were used with it. The event type can also distinguish double or triple button presses (see Section 22.7.7 [Repeat Events], page 447). The rest of the list elements give position and time information.

For key lookup, only the event type matters: two events of the same type necessarily run the same command. The command can access the full values of these events using the '`e`' interactive code. See Section 22.2.2 [Interactive Codes], page 428.

A key sequence that starts with a mouse event is read using the keymaps of the buffer in the window that the mouse was in, not the current buffer. This does not imply that clicking in a window selects that window or its buffer—that is entirely under the control of the command binding of the key sequence.

22.7.4 Click Events

When the user presses a mouse button and releases it at the same location, that generates a *click* event. Depending on how your window-system reports mouse-wheel events, turning the mouse wheel can generate either a mouse click or a mouse-wheel event. All mouse event share the same format:

```
(event-type position click-count)
```

event-type This is a symbol that indicates which mouse button was used. It is one of the symbols `mouse-1`, `mouse-2`, ..., where the buttons are numbered left to right. For mouse-wheel event, it can be `wheel-up` or `wheel-down`.

You can also use prefixes '`A-`', '`C-`', '`H-`', '`M-`', '`S-`' and '`s-`' for modifiers alt, control, hyper, meta, shift and super, just as you would with function keys.

This symbol also serves as the event type of the event. Key bindings describe events by their types; thus, if there is a key binding for `mouse-1`, that binding would apply to all events whose *event-type* is `mouse-1`.

position This is a *mouse position list* specifying where the mouse event occurred; see below for details.

click-count

This is the number of rapid repeated presses so far of the same mouse button or the number of repeated turns of the wheel. See Section 22.7.7 [Repeat Events], page 447.

To access the contents of a mouse position list in the *position* slot of a mouse event, you should typically use the functions documented in Section 22.7.15 [Accessing Mouse], page 457.

The explicit format of the list depends on where the event occurred. For clicks in the text area, mode line, header line, tab line, or in the fringe or marginal areas, the mouse position list has the form

```
(window pos-or-area (x . y) timestamp
 object text-pos (col . row)
 image (dx . dy) (width . height))
```

The meanings of these list elements are as follows:

window The window in which the mouse event occurred.

pos-or-area

The buffer position of the character clicked on in the text area; or, if the event was outside the text area, the window area where it occurred. It is one of the symbols `mode-line`, `header-line`, `tab-line`, `vertical-line`, `left-margin`, `right-margin`, `left-fringe`, or `right-fringe`.

In one special case, *pos-or-area* is a list containing a symbol (one of the symbols listed above) instead of just the symbol. This happens after the imaginary prefix keys for the event are registered by Emacs. See Section 22.8.1 [Key Sequence Input], page 462.

x, *y* The relative pixel coordinates of the event. For events in the text area of a window, the coordinate origin `(0 . 0)` is taken to be the top left corner of the text area. See Section 29.4 [Window Sizes], page 701. For events in a mode line, header line or tab line, the coordinate origin is the top left corner of the window itself. For fringes, margins, and the vertical border, *x* does not have meaningful data. For fringes and margins, *y* is relative to the bottom edge of the header line. In all cases, the *x* and *y* coordinates increase rightward and downward respectively.

timestamp

The time at which the event occurred, as an integer number of milliseconds since a system-dependent initial time.

object Either `nil`, which means the event occurred on buffer text, or a cons cell of the form (*string* . *string-pos*) if there is a string from a text property or an overlay at the event position.

string The string which was clicked on, including any properties.

string-pos The position in the string where the click occurred.

text-pos For clicks on a marginal area or on a fringe, this is the buffer position of the first visible character in the corresponding line in the window. For clicks on the mode line, the header line or the tab line, this is `nil`. For other events, it is the buffer position closest to the click.

col, *row* These are the actual column and row coordinate numbers of the glyph under the *x*, *y* position. If *x* lies beyond the last column of actual text on its line, *col* is reported by adding fictional extra columns that have the default character width. Row 0 is taken to be the header line if the window has one, or Row 1 if the window also has the tab line, or the topmost row of the text area otherwise. Column 0 is taken to be the leftmost column of the text area for clicks on a window text area, or the leftmost mode line or header line column for clicks there. For clicks on fringes or vertical borders, these have no meaningful data. For clicks on margins, *col* is measured from the left edge of the margin area and *row* is measured from the top of the margin area.

image If there is an image at the click location, this is the image object as returned by `find-image` (see Section 41.17.8 [Defining Images], page 1225); otherwise this is `nil`.

dx, *dy* These are the pixel offsets of the click relative to the top left corner of the *object*'s glyph that is the nearest one to the click. The relevant *object*s can be either a buffer, or a string, or an image, see above. If *object* is `nil` or a string, the coordinates are relative to the top left corner of the character glyph clicked on. Note that the offsets are always zero on text-mode frames, when *object* is `nil`, since each glyph there is considered to have exactly 1x1 pixel dimensions.

width, *height*
If the click is on a character, either from buffer text or from overlay or display string, these are the pixel width and height of that character's glyph; otherwise they are dimensions of *object* clicked on.

For clicks on a scroll bar, *position* has this form:

```
(window area (portion . whole) timestamp part)
```

window The window whose scroll bar was clicked on.

area This is the symbol `vertical-scroll-bar`.

portion The number of pixels from the top of the scroll bar to the click position. On some toolkits, including GTK+, Emacs cannot extract this data, so the value is always `0`.

whole The total length, in pixels, of the scroll bar. On some toolkits, including GTK+, Emacs cannot extract this data, so the value is always `0`.

timestamp
: The time at which the event occurred, in milliseconds. On some toolkits, including GTK+, Emacs cannot extract this data, so the value is always `0`.

part
: The part of the scroll bar on which the click occurred. It is one of the symbols `handle` (the scroll bar handle), `above-handle` (the area above the handle), `below-handle` (the area below the handle), `up` (the up arrow at one end of the scroll bar), or `down` (the down arrow at one end of the scroll bar).

For clicks on the frame's internal border (see Section 30.3.1 [Frame Layout], page 793), the frame's tool bar (see Section 23.18.6 [Tool Bar], page 518) or tab bar, *position* has this form:

```
(frame part (X . Y) timestamp)
```

frame
: The frame whose internal border or tool bar or tab bar was clicked on.

part
: The part of the frame which was clicked on. This can be one of the following:

  `tool-bar`
  : The frame has a tool bar, and the event was in the tool-bar area.

  `tab-bar`
  : The frame has a tab bar, and the event was in the tab-bar area.

  `left-edge`
  `top-edge`
  `right-edge`
  `bottom-edge`
  : The click was on the corresponding border at an offset of at least one canonical character from the border's nearest corner.

  `top-left-corner`
  `top-right-corner`
  `bottom-right-corner`
  `bottom-left-corner`
  : The click was on the corresponding corner of the internal border.

  `nil`
  : The frame does not have an internal border, and the event was not on the tab bar or the tool bar. This usually happens on text-mode frames. This can also happen on GUI frames with internal border if the frame doesn't have its `drag-internal-border` parameter (see Section 30.4.3.7 [Mouse Dragging Parameters], page 816) set to a non-`nil` value.

22.7.5 Drag Events

With Emacs, you can have a drag event without even changing your clothes. A *drag event* happens every time the user presses a mouse button and then moves the mouse to a different character position before releasing the button. Like all mouse events, drag events are represented in Lisp as lists. The lists record both the starting mouse position and the final position, like this:

```
(event-type
 (window1 START-POSITION)
 (window2 END-POSITION))
```

For a drag event, the name of the symbol *event-type* contains the prefix '`drag-`'. For example, dragging the mouse with button 2 held down generates a `drag-mouse-2` event. The second and third elements of the event give the starting and ending position of the drag, as mouse position lists (see Section 22.7.4 [Click Events], page 443). You can access the second element of any mouse event in the same way. However, the drag event may end outside the boundaries of the frame that was initially selected. In that case, the third element's position list contains that frame in place of a window.

The '`drag-`' prefix follows the modifier key prefixes such as '`C-`' and '`M-`'.

If `read-key-sequence` receives a drag event that has no key binding, and the corresponding click event does have a binding, it changes the drag event into a click event at the drag's starting position. This means that you don't have to distinguish between click and drag events unless you want to.

22.7.6 Button-Down Events

Click and drag events happen when the user releases a mouse button. They cannot happen earlier, because there is no way to distinguish a click from a drag until the button is released.

If you want to take action as soon as a button is pressed, you need to handle *button-down* events.[2] These occur as soon as a button is pressed. They are represented by lists that look exactly like click events (see Section 22.7.4 [Click Events], page 443), except that the *event-type* symbol name contains the prefix '`down-`'. The '`down-`' prefix follows modifier key prefixes such as '`C-`' and '`M-`'.

The function `read-key-sequence` ignores any button-down events that don't have command bindings; therefore, the Emacs command loop ignores them too. This means that you need not worry about defining button-down events unless you want them to do something. The usual reason to define a button-down event is so that you can track mouse motion (by reading motion events) until the button is released. See Section 22.7.8 [Motion Events], page 448.

22.7.7 Repeat Events

If you press the same mouse button more than once in quick succession without moving the mouse, Emacs generates special *repeat* mouse events for the second and subsequent presses.

The most common repeat events are *double-click* events. Emacs generates a double-click event when you click a button twice; the event happens when you release the button (as is normal for all click events).

The event type of a double-click event contains the prefix '`double-`'. Thus, a double click on the second mouse button with `meta` held down comes to the Lisp program as `M-double-mouse-2`. If a double-click event has no binding, the binding of the corresponding ordinary click event is used to execute it. Thus, you need not pay attention to the double click feature unless you really want to.

When the user performs a double click, Emacs generates first an ordinary click event, and then a double-click event. Therefore, you must design the command binding of the double click event to assume that the single-click command has already run. It must produce the desired results of a double click, starting from the results of a single click.

[2] Button-down is the conservative antithesis of drag.

This is convenient, if the meaning of a double click somehow builds on the meaning of a single click—which is recommended user interface design practice for double clicks.

If you click a button, then press it down again and start moving the mouse with the button held down, then you get a *double-drag* event when you ultimately release the button. Its event type contains '`double-drag`' instead of just '`drag`'. If a double-drag event has no binding, Emacs looks for an alternate binding as if the event were an ordinary drag.

Before the double-click or double-drag event, Emacs generates a *double-down* event when the user presses the button down for the second time. Its event type contains '`double-down`' instead of just '`down`'. If a double-down event has no binding, Emacs looks for an alternate binding as if the event were an ordinary button-down event. If it finds no binding that way either, the double-down event is ignored.

To summarize, when you click a button and then press it again right away, Emacs generates a down event and a click event for the first click, a double-down event when you press the button again, and finally either a double-click or a double-drag event.

If you click a button twice and then press it again, all in quick succession, Emacs generates a *triple-down* event, followed by either a *triple-click* or a *triple-drag*. The event types of these events contain '`triple`' instead of '`double`'. If any triple event has no binding, Emacs uses the binding that it would use for the corresponding double event.

If you click a button three or more times and then press it again, the events for the presses beyond the third are all triple events. Emacs does not have separate event types for quadruple, quintuple, etc. events. However, you can look at the event list to find out precisely how many times the button was pressed.

`event-click-count` *event* [Function]
: This function returns the number of consecutive button presses that led up to *event*. If *event* is a double-down, double-click or double-drag event, the value is 2. If *event* is a triple event, the value is 3 or greater. If *event* is an ordinary mouse event (not a repeat event), the value is 1.

`double-click-fuzz` [User Option]
: To generate repeat events, successive mouse button presses must be at approximately the same screen position. The value of `double-click-fuzz` specifies the maximum number of pixels the mouse may be moved (horizontally or vertically) between two successive clicks to make a double-click.

  This variable is also the threshold for motion of the mouse to count as a drag.

`double-click-time` [User Option]
: To generate repeat events, the number of milliseconds between successive button presses must be less than the value of `double-click-time`. Setting `double-click-time` to `nil` disables multi-click detection entirely. Setting it to `t` removes the time limit; Emacs then detects multi-clicks by position only.

22.7.8 Motion Events

Emacs sometimes generates *mouse motion* events to describe motion of the mouse without any button activity. Mouse motion events are represented by lists that look like this:

```
(mouse-movement POSITION)
```

position is a mouse position list (see Section 22.7.4 [Click Events], page 443), specifying the current position of the mouse cursor. As with the end-position of a drag event, this position list may represent a location outside the boundaries of the initially selected frame, in which case the list contains that frame in place of a window.

The `track-mouse` macro enables generation of motion events within its body. Outside of `track-mouse` body, Emacs does not generate events for mere motion of the mouse, and these events do not appear. See Section 30.15 [Mouse Tracking], page 837.

`mouse-fine-grained-tracking` [Variable]

When non-`nil`, mouse motion events are generated even for very small movements. Otherwise, motion events are not generated as long as the mouse cursor remains pointing to the same glyph in the text.

22.7.9 Touchscreen Events

Some window systems provide support for input devices that react to the user's touching the screen and moving fingers while touching the screen. These input devices are known as touchscreens, and Emacs reports the events they generate as *touchscreen events*.

Most individual events generated by a touchscreen only have meaning as part of a larger sequence of other events: for instance, the simple operation of tapping the touchscreen involves the user placing and raising a finger on the touchscreen, and swiping the display to scroll it involves placing a finger, moving it many times upwards or downwards, and then raising the finger.

While a simplistic model consisting of one finger is adequate for taps and scrolling, more complicated gestures require support for keeping track of multiple fingers, where the position of each finger is represented by a *touch point*. For example, a "pinch to zoom" gesture might consist of the user placing two fingers and moving them individually in opposite directions, where the distance between the positions of their individual points determine the amount by which to zoom the display, and the center of an imaginary line between those positions determines where to pan the display after zooming.

The low-level touchscreen events described below can be used to implement all the touch sequences described above. In those events, each point is represented by a cons of an arbitrary number identifying the point and a mouse position list (see Section 22.7.4 [Click Events], page 443) specifying the position of the finger when the event occurred.

`(touchscreen-begin` *`point`*`)`

This event is sent when *point* is created by the user pressing a finger against the touchscreen.

`(touchscreen-update` *`points`*`)`

This event is sent when a point on the touchscreen has changed position. *points* is a list of touch points containing the up-to-date positions of each touch point currently on the touchscreen.

`(touchscreen-end` *`point`*`)`

This event is sent when *point* is no longer present on the display, because another program took the grab, or because the user raised the finger from the touchscreen.

22.7.10 Focus Events

Window systems provide general ways for the user to control which window gets keyboard input. This choice of window is called the *focus*. When the user does something to switch between Emacs frames, that generates a *focus event*. The normal definition of a focus event, in the global keymap, is to select a new frame within Emacs, as the user would expect. See Section 30.10 [Input Focus], page 827, which also describes hooks related to focus events.

Focus events are represented in Lisp as lists that look like this:

```
(switch-frame new-frame)
```

where *new-frame* is the frame switched to.

Some X window managers are set up so that just moving the mouse into a window is enough to set the focus there. Usually, there is no need for a Lisp program to know about the focus change until some other kind of input arrives. Emacs generates a focus event only when the user actually types a keyboard key or presses a mouse button in the new frame; just moving the mouse between frames does not generate a focus event.

A focus event in the middle of a key sequence would garble the sequence. So Emacs never generates a focus event in the middle of a key sequence. If the user changes focus in the middle of a key sequence—that is, after a prefix key—then Emacs reorders the events so that the focus event comes either before or after the multi-event key sequence, and not within it.

22.7.11 Xwidget events

Xwidgets (see Section 41.19 [Xwidgets], page 1233) can send events to update Lisp programs on their status. These events are dubbed `xwidget-events`, and contain various data describing the nature of the change.

`(xwidget-event` *kind xwidget arg*`)`

This event is sent whenever some kind of update occurs in *xwidget*. There are several types of updates, identified by their *kind*.

It is a special event (see Section 22.9 [Special Events], page 471), which should be handled by adding a callback to an xwidget that is called whenever an xwidget event for *xwidget* is received.

You can add a callback by setting the `callback` of an xwidget's property list, which should be a function that accepts *xwidget* and *kind* as arguments.

`load-changed`

This xwidget event indicates that the *xwidget* has reached a particular point of the page-loading process. When these events are sent, *arg* will contain a string that further describes the status of the widget:

'`load-started`'

This means the widget has begun a page-loading operation.

'`load-finished`'

This means the *xwidget* has finished processing whatever page-loading operation that it was previously performing.

‘`load-redirected`’
: This means the *xwidget* has encountered and followed a redirect during the page-loading operation.

‘`load-committed`’
: This means the *xwidget* has committed to a given URL during the page-loading operation, i.e. the URL is the final URL that will be rendered by *xwidget* during the current page-loading operation.

`download-callback`
: This event indicates that a download of some kind has been completed.

In the above events, there can be arguments after *arg*, which itself indicates the URL from which the download file was retrieved: the first argument after *arg* indicates the MIME type of the download, as a string, while the second argument contains the full file name of the downloaded file.

`download-started`
: This event indicates that a download has been started. In these events, *arg* contains the URL of the file that is currently being downloaded.

`javascript-callback`
: This event contains JavaScript callback data. These events are used internally by `xwidget-webkit-execute-script`.

`(xwidget-display-event` *xwidget source*`)`
: This event is sent whenever an xwidget requests that another xwidget be displayed. *xwidget* is the xwidget that should be displayed, and *source* is the xwidget that asked to display *xwidget*.

  It is also a special event which should be handled through callbacks. You can add such a callback by setting the `display-callback` of *source*'s property list, which should be a function that accepts *xwidget* and *source* as arguments.

  xwidget's buffer will be set to a temporary buffer. When displaying the widget, care should be taken to replace the buffer with the buffer in which the xwidget will be displayed, using `set-xwidget-buffer` (see Section 41.19 [Xwidgets], page 1233).

22.7.12 Miscellaneous System Events

A few other event types represent occurrences within the system.

`(delete-frame (`*frame*`))`
: This kind of event indicates that the user gave the window manager a command to delete a particular window, which happens to be an Emacs frame.

  The standard definition of the `delete-frame` event is to delete *frame*.

`(iconify-frame (`*frame*`))`
: This kind of event indicates that the user iconified *frame* using the window manager. Its standard definition is `ignore`; since the frame has already been

iconified, Emacs has no work to do. The purpose of this event type is so that you can keep track of such events if you want to.

`(make-frame-visible (`*frame*`))`
: This kind of event indicates that the user deiconified *frame* using the window manager. Its standard definition is `ignore`; since the frame has already been made visible, Emacs has no work to do.

`(touch-end (`*position*`))`
: This kind of event indicates that the user's finger moved off the mouse wheel or the touchpad. The *position* element is a mouse position list (see Section 22.7.4 [Click Events], page 443), specifying the position of the mouse cursor when the finger moved off the mouse wheel.

`(wheel-up` *position clicks lines pixel-delta*`)`
`(wheel-down` *position clicks lines pixel-delta*`)`
: These events are generated by moving a mouse wheel. The *position* element is a mouse position list (see Section 22.7.4 [Click Events], page 443), specifying the position of the mouse cursor when the event occurred.

  clicks, if present, is the number of times that the wheel was moved in quick succession. See Section 22.7.7 [Repeat Events], page 447. *lines*, if present and not `nil`, is the positive number of screen lines that should be scrolled (either up, when the event is `wheel-up`, or down when the event is `wheel-down`). *pixel-delta*, if present, is a cons cell of the form `(`*x* `.` *y*`)`, where *x* and *y* are the numbers of pixels by which to scroll in each axis, a.k.a. *pixelwise deltas*. Usually, only one of the two will be non-zero, the other will be either zero or very close to zero; the larger number indicates the axis to scroll the window. When the variable `mwheel-coalesce-scroll-events` is `nil`, the scroll commands ignore the *lines* element, even if it's non-`nil`, and use the *pixel-delta* data instead; in that case, the direction of scrolling is determined by the sign of the pixelwise deltas, and the direction (up or down) implied by the event kind is ignored.

  You can use these *x* and *y* pixelwise deltas to determine how much the mouse wheel has actually moved at pixel resolution. For example, the pixelwise deltas could be used to scroll the display at pixel resolution, exactly according to the user's turning the mouse wheel. This pixelwise scrolling is possible only when `mwheel-coalesce-scroll-events` is `nil`, and in general the *pixel-delta* data is not generated when that variable is non-`nil`.

  The `wheel-up` and `wheel-down` events are generated only on some kinds of systems. On other systems, `mouse-4` and `mouse-5` are used instead. For portable code, use the variables `mouse-wheel-up-event`, `mouse-wheel-up-alternate-event`, `mouse-wheel-down-event` and `mouse-wheel-down-alternate-event` defined in `mwheel.el` to determine what event types to expect from the mouse wheel.

  Similarly, some mice can generate `mouse-wheel-left-event` and `mouse-wheel-right-event` and can be used to scroll if `mouse-wheel-tilt-scroll` is non-`nil`. However, some mice also generate other events at the same time as they're generating these scroll events which may get in the way. The

way to fix this is generally to unbind these events (for instance, `mouse-6` or `mouse-7`, but this is very hardware and operating system dependent).

`(pinch` *position dx dy scale angle*`)`

This kind of event is generated by the user performing a "pinch" gesture by placing two fingers on a touchpad and moving them towards or away from each other. *position* is a mouse position list (see Section 22.7.4 [Click Events], page 443) that provides the position of the mouse pointer when the event occurred, *dx* is the change in the horizontal distance between the fingers since the last event in the same sequence, *dy* is the vertical movement of the fingers since the last event in the same sequence, *scale* is the ratio of the current distance between the fingers to that distance at the start of the sequence, and *angle* is the angular difference in degrees between the direction of the line connecting the fingers in this event and the direction of that line in the last event of the same sequence.

As pinch events are only sent at the beginning or during a pinch sequence, they do not report gestures where the user moves two fingers on a touchpad in a rotating fashion without pinching the fingers.

All arguments after *position* are floating point numbers.

This event is usually sent as part of a sequence, which begins with the user placing two fingers on the touchpad, and ends with the user removing those fingers. *dx*, *dy*, and *angle* will be `0.0` in the first event of a sequence; subsequent events will report non-zero values for these members of the event structure.

dx and *dy* are reported in imaginary relative units, in which `1.0` is the width and height of the touchpad respectively. They are usually interpreted as being relative to the size of the object beneath the gesture: image, window, etc.

`(preedit-text` *arg*`)`

This event is sent when a system input method tells Emacs to display some text to indicate to the user what will be inserted. The contents of *arg* are dependent on the window system being used.

On X, *arg* is a string describing some text to place behind the cursor. It can be `nil`, which means to remove any text previously displayed.

On PGTK frames (see Chapter 30 [Frames], page 787), *arg* is a list of strings with information about their color and underline attributes. It has the following form:

```
((string1
  (ul . underline-color)
  (bg . background-color)
  (fg . foreground-color))
 (string2
  (ul . underline-color)
  (bg . background-color)
  (fg . foreground-color))
 ...
)
```

Color information can be omitted, leaving just the text of the strings. *underline-color* can be `t`, meaning underlined text with default underline color, or it can be a string, the name of the color to draw the underline.

This is a special event (see Section 22.9 [Special Events], page 471), which normally should not be bound by the user to any command. Emacs will typically display the text contained in the event in an overlay behind point when it is received.

`(drag-n-drop` *position files*`)`

This kind of event is generated when a group of files is selected in an application outside of Emacs, and then dragged and dropped onto an Emacs frame.

The element *position* is a list describing the position of the event, in the same format as used in a mouse-click event (see Section 22.7.4 [Click Events], page 443), and *files* is the list of file names that were dragged and dropped. The usual way to handle this event is by visiting these files.

This kind of event is generated, at present, only on some kinds of systems.

`help-echo`

This kind of event is generated when a mouse pointer moves onto a portion of buffer text which has a `help-echo` text property. The generated event has this form:

```
(help-echo frame help window object pos)
```

The precise meaning of the event parameters and the way these parameters are used to display the help-echo text are described in [Text help-echo], page 928.

`sigusr1`
`sigusr2`

These events are generated when the Emacs process receives the signals `SIGUSR1` and `SIGUSR2`. They contain no additional data because signals do not carry additional information. They can be useful for debugging (see Section 19.1.1 [Error Debugging], page 334).

To catch a user signal, bind the corresponding event to an interactive command in the `special-event-map` (see Section 23.9 [Controlling Active Maps], page 492). The command is called with no arguments, and the specific signal event is available in `last-input-event` (see Section 22.8.6 [Event Input Misc], page 469. For example:

```
(defun sigusr-handler ()
  (interactive)
  (message "Caught signal %S" last-input-event))

(keymap-set special-event-map "<sigusr1>" 'sigusr-handler)
```

To test the signal handler, you can make Emacs send a signal to itself:

```
(signal-process (emacs-pid) 'sigusr1)
```

`language-change`

This kind of event is generated on MS-Windows when the input language has changed. This typically means that the keyboard keys will send to Emacs characters from a different language. The generated event has this form:

```
(language-change frame codepage language-id)
```

Here *frame* is the frame which was current when the input language changed; *codepage* is the new codepage number; and *language-id* is the numerical ID of the new input language. The coding-system (see Section 34.10 [Coding Systems], page 980) that corresponds to *codepage* is `cp`*codepage* or `windows-`*codepage*. To convert *language-id* to a string (e.g., to use it for various language-dependent features, such as `set-language-environment`), use the `w32-get-locale-info` function, like this:

```
;; Get the abbreviated language name, such as "ENU" for English
(w32-get-locale-info language-id)
;; Get the full English name of the language,
;; such as "English (United States)"
(w32-get-locale-info language-id 4097)
;; Get the full localized name of the language
(w32-get-locale-info language-id t)
```

`end-session`

This event is generated on MS-Windows when the operating system informs Emacs that the user terminated the interactive session, or that the system is shutting down. The standard definition of this event is to invoke the `kill-emacs` command (see Section 42.2.1 [Killing Emacs], page 1267) so as to shut down Emacs in an orderly fashion; if there are unsaved changes, this will produce auto-save files (see Section 27.2 [Auto-Saving], page 666) that the user can use after restarting the session to restore the unsaved edits.

If one of these events arrives in the middle of a key sequence—that is, after a prefix key—then Emacs reorders the events so that this event comes either before or after the multi-event key sequence, not within it.

Some of these special events, such as `delete-frame`, invoke Emacs commands by default; others are not bound. If you want to arrange for a special event to invoke a command, you can do that via `special-event-map`. The command you bind to a function key in that map can then examine the full event which invoked it in `last-input-event`. See Section 22.9 [Special Events], page 471.

22.7.13 Event Examples

If the user presses and releases the left mouse button over the same location, that generates a sequence of events like this:

```
(down-mouse-1 (#<window 18 on NEWS> 2613 (0 . 38) -864320))
(mouse-1      (#<window 18 on NEWS> 2613 (0 . 38) -864180))
```

While holding the control key down, the user might hold down the second mouse button, and drag the mouse from one line to the next. That produces two events, as shown here:

```
(C-down-mouse-2 (#<window 18 on NEWS> 3440 (0 . 27) -731219))
(C-drag-mouse-2 (#<window 18 on NEWS> 3440 (0 . 27) -731219)
                (#<window 18 on NEWS> 3510 (0 . 28) -729648))
```

While holding down the meta and shift keys, the user might press the second mouse button on the window's mode line, and then drag the mouse into another window. That produces a pair of events like these:

```
(M-S-down-mouse-2 (#<window 18 on NEWS> mode-line (33 . 31) -457844))
(M-S-drag-mouse-2 (#<window 18 on NEWS> mode-line (33 . 31) -457844)
                  (#<window 20 on carlton-sanskrit.tex> 161 (33 . 3)
```

```
                -453816))
```

The frame with input focus might not take up the entire screen, and the user might move the mouse outside the scope of the frame. Inside the `track-mouse` macro, that produces an event like this:

```
(mouse-movement (#<frame *ielm* 0x102849a30> nil (563 . 205) 532301936))
```

22.7.14 Classifying Events

Every event has an *event type*, which classifies the event for key binding purposes. For a keyboard event, the event type equals the event value; thus, the event type for a character is the character, and the event type for a function key symbol is the symbol itself. For events that are lists, the event type is the symbol in the CAR of the list. Thus, the event type is always a symbol or a character.

Two events of the same type are equivalent where key bindings are concerned; thus, they always run the same command. That does not necessarily mean they do the same things, however, as some commands look at the whole event to decide what to do. For example, some commands use the location of a mouse event to decide where in the buffer to act.

Sometimes broader classifications of events are useful. For example, you might want to ask whether an event involved the `META` key, regardless of which other key or mouse button was used.

The functions `event-modifiers` and `event-basic-type` are provided to get such information conveniently.

`event-modifiers` *event* [Function]

This function returns a list of the modifiers that *event* has. The modifiers are symbols; they include `shift`, `control`, `meta`, `alt`, `hyper` and `super`. In addition, the modifiers list of a mouse event symbol always contains one of `click`, `drag`, and `down`. For double or triple events, it also contains `double` or `triple`.

The argument *event* may be an entire event object, or just an event type. If *event* is a symbol that has never been used in an event that has been read as input in the current Emacs session, then `event-modifiers` can return `nil`, even when *event* actually has modifiers.

Here are some examples:

```
(event-modifiers ?a)
     ⇒ nil
(event-modifiers ?A)
     ⇒ (shift)
(event-modifiers ?\C-a)
     ⇒ (control)
(event-modifiers ?\C-%)
     ⇒ (control)
(event-modifiers ?\C-\S-a)
     ⇒ (control shift)
(event-modifiers 'f5)
     ⇒ nil
(event-modifiers 's-f5)
```

```
     ⇒ (super)
(event-modifiers 'M-S-f5)
     ⇒ (meta shift)
(event-modifiers 'mouse-1)
     ⇒ (click)
(event-modifiers 'down-mouse-1)
     ⇒ (down)
```

The modifiers list for a click event explicitly contains `click`, but the event symbol name itself does not contain '`click`'. Similarly, the modifiers list for an ASCII control character, such as '`C-a`', contains `control`, even though reading such an event via `read-char` will return the value 1 with the control modifier bit removed.

`event-basic-type` *event* [Function]

This function returns the key or mouse button that *event* describes, with all modifiers removed. The *event* argument is as in `event-modifiers`. For example:

```
(event-basic-type ?a)
     ⇒ 97
(event-basic-type ?A)
     ⇒ 97
(event-basic-type ?\C-a)
     ⇒ 97
(event-basic-type ?\C-\S-a)
     ⇒ 97
(event-basic-type 'f5)
     ⇒ f5
(event-basic-type 's-f5)
     ⇒ f5
(event-basic-type 'M-S-f5)
     ⇒ f5
(event-basic-type 'down-mouse-1)
     ⇒ mouse-1
```

`mouse-movement-p` *object* [Function]

This function returns non-`nil` if *object* is a mouse movement event. See Section 22.7.8 [Motion Events], page 448.

22.7.15 Accessing Mouse Events

This section describes convenient functions for accessing the data in a mouse button or motion event. Keyboard event data can be accessed using the same functions, but data elements that aren't applicable to keyboard events are zero or `nil`.

The following two functions return a mouse position list (see Section 22.7.4 [Click Events], page 443), specifying the position of a mouse event.

`event-start` *event* [Function]

This returns the starting position of *event*.

If *event* is a click or button-down event, this returns the location of the event. If *event* is a drag event, this returns the drag's starting position.

`event-end` *event* [Function]
This returns the ending position of *event*.

If *event* is a drag event, this returns the position where the user released the mouse button. If *event* is a click or button-down event, the value is actually the starting position, which is the only position such events have.

`posnp` *object* [Function]
This function returns non-`nil` if *object* is a mouse position list, in the format documented in Section 22.7.4 [Click Events], page 443); and `nil` otherwise.

These functions take a mouse position list as argument, and return various parts of it:

`posn-window` *position* [Function]
Return the window that *position* is in. If *position* represents a location outside the frame where the event was initiated, return that frame instead.

`posn-area` *position* [Function]
Return the window area recorded in *position*. It returns `nil` when the event occurred in the text area of the window; otherwise, it is a symbol identifying the area in which the event occurred.

`posn-point` *position* [Function]
Return the buffer position in *position*. When the event occurred in the text area of the window, in a marginal area, or on a fringe, this is an integer specifying a buffer position. Otherwise, the value is undefined.

`posn-x-y` *position* [Function]
Return the pixel-based x and y coordinates in *position*, as a cons cell (*x* . *y*). These coordinates are relative to the window given by `posn-window`.

This example shows how to convert the window-relative coordinates in the text area of a window into frame-relative coordinates:

```
(defun frame-relative-coordinates (position)
  "Return frame-relative coordinates from POSITION.
POSITION is assumed to lie in a window text area."
  (let* ((x-y (posn-x-y position))
         (window (posn-window position))
         (edges (window-inside-pixel-edges window)))
    (cons (+ (car x-y) (car edges))
          (+ (cdr x-y) (cadr edges)))))
```

`posn-col-row` *position* **&optional** *use-window* [Function]
This function returns a cons cell (*col* . *row*), containing the estimated column and row corresponding to buffer position described by *position*. The return value is given in units of the frame's default character width and default line height (including spacing), as computed from the *x* and *y* values corresponding to *position*. (So, if the actual characters have non-default sizes, the actual row and column may differ from these computed values.) If the optional *window* argument is non-`nil`, use the default character width in the window indicated by *position* instead of the frame. (This

makes a difference if that window is showing a buffer with a non-default zooming level, for instance.)

Note that *row* is counted from the top of the text area. If the window given by *position* possesses a header line (see Section 24.4.7 [Header Lines], page 560) or a tab line, they are *not* included in the *row* count.

`posn-actual-col-row` *position* [Function]
Return the actual row and column in *position*, as a cons cell `(col . row)`. The values are the actual row and column numbers in the window given by *position*. See Section 22.7.4 [Click Events], page 443, for details. The function returns `nil` if *position* does not include actual position values; in that case `posn-col-row` can be used to get approximate values.

Note that this function doesn't account for the visual width of characters on display, like the number of visual columns taken by a tab character or an image. If you need the coordinates in canonical character units, use `posn-col-row` instead.

`posn-string` *position* [Function]
Return the string object described by *position*, either `nil` (which means *position* describes buffer text), or a cons cell `(string . string-pos)`.

`posn-image` *position* [Function]
Return the image object in *position*, either `nil` (if there's no image at *position*), or an image spec `(image ...)`.

`posn-object` *position* [Function]
Return the image or string object described by *position*, either `nil` (which means *position* describes buffer text), an image `(image ...)`, or a cons cell `(string . string-pos)`.

`posn-object-x-y` *position* [Function]
Return the pixel-based x and y coordinates relative to the upper left corner of the object described by *position*, as a cons cell `(dx . dy)`. If the *position* describes buffer text, return the relative coordinates of the buffer-text character closest to that position.

`posn-object-width-height` *position* [Function]
Return the pixel width and height of the object described by *position*, as a cons cell `(width . height)`. If the *position* describes a buffer position, return the size of the character at that position.

`posn-timestamp` *position* [Function]
Return the timestamp in *position*. This is the time at which the event occurred, in milliseconds. Such a timestamp is reported relative to an arbitrary starting time that varies according to the window system in use. On the X Window System, for example, it is the number of milliseconds since the X server was started.

These functions compute a position list given particular buffer position or screen position. You can access the data in this position list with the functions described above.

`posn-at-point` **&optional** *pos window* [Function]
This function returns a position list for position *pos* in *window*. *pos* defaults to point in *window*; *window* defaults to the selected window.

`posn-at-point` returns `nil` if *pos* is not visible in *window*.

`posn-at-x-y` *x y* **&optional** *frame-or-window whole* [Function]
This function returns position information corresponding to pixel coordinates *x* and *y* in a specified frame or window, *frame-or-window*, which defaults to the selected window. The coordinates *x* and *y* are relative to the text area of the selected window. If *whole* is non-`nil`, the *x* coordinate is relative to the entire window area including scroll bars, margins and fringes.

22.7.16 Accessing Scroll Bar Events

These functions are useful for decoding scroll bar events.

`scroll-bar-event-ratio` *event* [Function]
This function returns the fractional vertical position of a scroll bar event within the scroll bar. The value is a cons cell (*portion* . *whole*) containing two integers whose ratio is the fractional position.

`scroll-bar-scale` *ratio total* [Function]
This function multiplies (in effect) *ratio* by *total*, rounding the result to an integer. The argument *ratio* is not a number, but rather a pair (*num* . *denom*)—typically a value returned by `scroll-bar-event-ratio`.

This function is handy for scaling a position on a scroll bar into a buffer position. Here's how to do that:

```
(+ (point-min)
   (scroll-bar-scale
      (posn-x-y (event-start event))
      (- (point-max) (point-min))))
```

Recall that scroll bar events have two integers forming a ratio, in place of a pair of x and y coordinates.

22.7.17 Putting Keyboard Events in Strings

In most of the places where strings are used, we conceptualize the string as containing text characters—the same kind of characters found in buffers or files. Occasionally Lisp programs use strings that conceptually contain keyboard characters; for example, they may be key sequences or keyboard macro definitions. However, storing keyboard characters in a string is a complex matter, for reasons of historical compatibility, and it is not always possible.

We recommend that new programs avoid dealing with these complexities by not storing keyboard events in strings containing control characters or the like, but instead store them in the common Emacs format as understood by `key-valid-p`.

If you read a key sequence with `read-key-sequence-vector` (or `read-key-sequence`), or access a key sequence with `this-command-keys-vector` (or `this-command-keys`), you can transform this to the recommended format by using `key-description`.

The complexities stem from the modifier bits that keyboard input characters can include. Aside from the Meta modifier, none of these modifier bits can be included in a string, and the Meta modifier is allowed only in special cases.

The earliest GNU Emacs versions represented meta characters as codes in the range of 128 to 255. At that time, the basic character codes ranged from 0 to 127, so all keyboard character codes did fit in a string. Many Lisp programs used '`\M-`' in string constants to stand for meta characters, especially in arguments to `define-key` and similar functions, and key sequences and sequences of events were always represented as strings.

When we added support for larger basic character codes beyond 127, and additional modifier bits, we had to change the representation of meta characters. Now the flag that represents the Meta modifier in a character is 2^{27} and such numbers cannot be included in a string.

To support programs with '`\M-`' in string constants, there are special rules for including certain meta characters in a string. Here are the rules for interpreting a string as a sequence of input characters:

- If the keyboard character value is in the range of 0 to 127, it can go in the string unchanged.
- The meta variants of those characters, with codes in the range of 2^{27} to $2^{27} + 127$, can also go in the string, but you must change their numeric values. You must set the 2^7 bit instead of the 2^{27} bit, resulting in a value between 128 and 255. Only a unibyte string can include these codes.
- Non-ASCII characters above 256 can be included in a multibyte string.
- Other keyboard character events cannot fit in a string. This includes keyboard events in the range of 128 to 255.

Functions such as `read-key-sequence` that construct strings of keyboard input characters follow these rules: they construct vectors instead of strings, when the events won't fit in a string.

When you use the read syntax '`\M-`' in a string, it produces a code in the range of 128 to 255—the same code that you get if you modify the corresponding keyboard event to put it in the string. Thus, meta events in strings work consistently regardless of how they get into the strings.

However, most programs would do well to avoid these issues by following the recommendations at the beginning of this section.

22.8 Reading Input

The editor command loop reads key sequences using the function `read-key-sequence`, which uses `read-event`. These and other functions for event input are also available for use in Lisp programs. See also `momentary-string-display` in Section 41.8 [Temporary Displays], page 1150, and `sit-for` in Section 22.10 [Waiting], page 471. See Section 42.13 [Terminal Input], page 1290, for functions and variables for controlling terminal input modes and debugging terminal input.

For higher-level input facilities, see Chapter 21 [Minibuffers], page 386.

22.8.1 Key Sequence Input

The command loop reads input a key sequence at a time, by calling `read-key-sequence`. Lisp programs can also call this function; for example, `describe-key` uses it to read the key to describe.

`read-key-sequence` *prompt* **&optional** *continue-echo* [Function]
dont-downcase-last switch-frame-ok command-loop

This function reads a key sequence and returns it as a string or vector. It keeps reading events until it has accumulated a complete key sequence; that is, enough to specify a non-prefix command using the currently active keymaps. (Remember that a key sequence that starts with a mouse event is read using the keymaps of the buffer in the window that the mouse was in, not the current buffer.)

If the events are all characters and all can fit in a string, then `read-key-sequence` returns a string (see Section 22.7.17 [Strings of Events], page 460). Otherwise, it returns a vector, since a vector can hold all kinds of events—characters, symbols, and lists. The elements of the string or vector are the events in the key sequence.

Reading a key sequence includes translating the events in various ways. See Section 23.15 [Translation Keymaps], page 505.

The argument *prompt* is either a string to be displayed in the echo area as a prompt, or `nil`, meaning not to display a prompt. The argument *continue-echo*, if non-`nil`, means to echo this key as a continuation of the previous key.

Normally any upper case event is converted to lower case if the original event is undefined and the lower case equivalent is defined. The argument *dont-downcase-last*, if non-`nil`, means do not convert the last event to lower case. This is appropriate for reading a key sequence to be defined.

The argument *switch-frame-ok*, if non-`nil`, means that this function should process a `switch-frame` event if the user switches frames before typing anything. If the user switches frames in the middle of a key sequence, or at the start of the sequence but *switch-frame-ok* is `nil`, then the event will be put off until after the current key sequence.

The argument *command-loop*, if non-`nil`, means that this key sequence is being read by something that will read commands one after another. It should be `nil` if the caller will read just one key sequence.

In the following example, Emacs displays the prompt '`?`' in the echo area, and then the user types *C-x C-f*.

```
(read-key-sequence "?")

---------- Echo Area ----------
?C-x C-f
---------- Echo Area ----------

     ⇒ "^X^F"
```

The function `read-key-sequence` suppresses quitting: *C-g* typed while reading with this function works like any other character, and does not set `quit-flag`. See Section 22.11 [Quitting], page 472.

`read-key-sequence-vector` *prompt* **&optional** *continue-echo dont-downcase-last switch-frame-ok command-loop* [Function]

This is like `read-key-sequence` except that it always returns the key sequence as a vector, never as a string. See Section 22.7.17 [Strings of Events], page 460.

If an input character is upper-case (or has the shift modifier) and has no key binding, but its lower-case equivalent has one, then `read-key-sequence` converts the character to lower case. (This behavior can be disabled by setting the `translate-upper-case-key-bindings` user option to `nil`.) Note that `lookup-key` does not perform case conversion in this way.

When reading input results in such a *shift-translation*, Emacs sets the variable `this-command-keys-shift-translated` to a non-`nil` value. Lisp programs can examine this variable if they need to modify their behavior when invoked by shift-translated keys. For example, the function `handle-shift-selection` examines the value of this variable to determine how to activate or deactivate the region (see Section 32.7 [The Mark], page 875).

The function `read-key-sequence` also transforms some mouse events. It converts unbound drag events into click events, and discards unbound button-down events entirely. It also reshuffles focus events and miscellaneous window events so that they never appear in a key sequence with any other events.

When mouse events occur in special parts of a window or frame, such as a mode line or a scroll bar, the event type shows nothing special—it is the same symbol that would normally represent that combination of mouse button and modifier keys. The information about the window part is kept elsewhere in the event—in the coordinates. But `read-key-sequence` translates this information into imaginary prefix keys, all of which are symbols: `tab-line`, `header-line`, `horizontal-scroll-bar`, `menu-bar`, `tab-bar`, `mode-line`, `vertical-line`, `vertical-scroll-bar`, `left-margin`, `right-margin`, `left-fringe`, `right-fringe`, `right-divider`, and `bottom-divider`. You can define meanings for mouse clicks in special window parts by defining key sequences using these imaginary prefix keys.

For example, if you call `read-key-sequence` and then click the mouse on the window's mode line, you get two events, like this:

```
(read-key-sequence "Click on the mode line: ")
     ⇒ [mode-line
        (mouse-1
         (#<window 6 on NEWS> mode-line
          (40 . 63) 5959987))]
```

`num-input-keys` [Variable]

This variable's value is the number of key sequences processed so far in this Emacs session. This includes key sequences read from the terminal and key sequences read from keyboard macros being executed.

22.8.2 Reading One Event

The lowest level functions for command input are `read-event`, `read-char`, and `read-char-exclusive`.

If you need a function to read a character using the minibuffer, use `read-char-from-minibuffer` (see Section 21.8 [Multiple Queries], page 416).

`read-event` **&optional** *prompt inherit-input-method seconds* [Function]
This function reads and returns the next event of command input, waiting if necessary until an event is available.

The returned event may come directly from the user, or from a keyboard macro. It is not decoded by the keyboard's input coding system (see Section 34.10.8 [Terminal I/O Encoding], page 993).

If the optional argument *prompt* is non-`nil`, it should be a string to display in the echo area as a prompt. If *prompt* is `nil` or the string '`""`', `read-event` does not display any message to indicate it is waiting for input; instead, it prompts by echoing: it displays descriptions of the events that led to or were read by the current command. See Section 41.4 [The Echo Area], page 1135.

If *inherit-input-method* is non-`nil`, then the current input method (if any) is employed to make it possible to enter a non-ASCII character. Otherwise, input method handling is disabled for reading this event.

If `cursor-in-echo-area` is non-`nil`, then `read-event` moves the cursor temporarily to the echo area, to the end of any message displayed there. Otherwise `read-event` does not move the cursor.

If *seconds* is non-`nil`, it should be a number specifying the maximum time to wait for input, in seconds. If no input arrives within that time, `read-event` stops waiting and returns `nil`. A floating point *seconds* means to wait for a fractional number of seconds. Some systems support only a whole number of seconds; on these systems, *seconds* is rounded down. If *seconds* is `nil`, `read-event` waits as long as necessary for input to arrive.

If *seconds* is `nil`, Emacs is considered idle while waiting for user input to arrive. Idle timers—those created with `run-with-idle-timer` (see Section 42.12 [Idle Timers], page 1289)—can run during this period. However, if *seconds* is non-`nil`, the state of idleness remains unchanged. If Emacs is non-idle when `read-event` is called, it remains non-idle throughout the operation of `read-event`; if Emacs is idle (which can happen if the call happens inside an idle timer), it remains idle.

If `read-event` gets an event that is defined as a help character, then in some cases `read-event` processes the event directly without returning. See Section 25.6 [Help Functions], page 604. Certain other events, called *special events*, are also processed directly within `read-event` (see Section 22.9 [Special Events], page 471).

Here is what happens if you call `read-event` and then press the right-arrow function key:

```
(read-event)
     ⇒ right
```

`read-char` **&optional** *prompt inherit-input-method seconds* [Function]
This function reads and returns a character input event. If the user generates an event which is not a character (i.e., a mouse click or function key event), `read-char` signals an error. The arguments work as in `read-event`.

If the event has modifiers, Emacs attempts to resolve them and return the code of the corresponding character. For example, if the user types `C-a`, the function returns 1, which is the ASCII code of the '`C-a`' character. If some of the modifiers cannot

be reflected in the character code, `read-char` leaves the unresolved modifier bits set in the returned event. For example, if the user types *C-M-*`a`, the function returns 134217729, 8000001 in hex, i.e. '`C-a`' with the Meta modifier bit set. This value is not a valid character code: it fails the `characterp` test (see Section 34.5 [Character Codes], page 970). Use `event-basic-type` (see Section 22.7.14 [Classifying Events], page 456) to recover the character code with the modifier bits removed; use `event-modifiers` to test for modifiers in the character event returned by `read-char`.

In the first example below, the user types the character *1* (ASCII code 49). The second example shows a keyboard macro definition that calls `read-char` from the minibuffer using `eval-expression`. `read-char` reads the keyboard macro's very next character, which is *1*. Then `eval-expression` displays its return value in the echo area.

```
(read-char)
     ⇒ 49

;; We assume here you use M-: to evaluate this.
(symbol-function 'foo)
     ⇒ "^[:(read-char)^M1"
(execute-kbd-macro 'foo)
     ⊣ 49
     ⇒ nil
```

`read-char-exclusive` **&optional** *prompt inherit-input-method seconds* [Function]

This function reads and returns a character input event. If the user generates an event which is not a character event, `read-char-exclusive` ignores it and reads another event, until it gets a character. The arguments work as in `read-event`. The returned value may include modifier bits, as with `read-char`.

None of the above functions suppress quitting.

`num-nonmacro-input-events` [Variable]

This variable holds the total number of input events received so far from the terminal—not counting those generated by keyboard macros.

We emphasize that, unlike `read-key-sequence`, the functions `read-event`, `read-char`, and `read-char-exclusive` do not perform the translations described in Section 23.15 [Translation Keymaps], page 505. If you wish to read a single key taking these translations into account (for example, to read Section 22.7.2 [Function Keys], page 442, in a terminal or Section 22.7.3 [Mouse Events], page 443, from `xterm-mouse-mode`), use the function `read-key`:

`read-key` **&optional** *prompt disable-fallbacks* [Function]

This function reads a single key. It is intermediate between `read-key-sequence` and `read-event`. Unlike the former, it reads a single key, not a key sequence. Unlike the latter, it does not return a raw event, but decodes and translates the user input according to `input-decode-map`, `local-function-key-map`, and `key-translation-map` (see Section 23.15 [Translation Keymaps], page 505).

The argument *prompt* is either a string to be displayed in the echo area as a prompt, or `nil`, meaning not to display a prompt.

If argument *disable-fallbacks* is non-`nil` then the usual fallback logic for unbound keys in `read-key-sequence` is not applied. This means that mouse button-down and multi-click events will not be discarded and `local-function-key-map` and `key-translation-map` will not get applied. If `nil` or unspecified, the only fallback disabled is downcasing of the last event.

`read-char-choice` *prompt chars* **&optional** *inhibit-quit* [Function]

This function uses `read-from-minibuffer` to read and return a single character that is a member of *chars*, which should be a list of single characters. It discards any input characters that are not members of *chars*, and shows a message to that effect.

The optional argument *inhibit-quit* is by default ignored, but if the variable `read-char-choice-use-read-key` is non-`nil`, this function uses `read-key` instead of `read-from-minibuffer`, and in that case *inhibit-quit* non-`nil` means ignore keyboard-quit events while waiting for valid input. In addition, if `read-char-choice-use-read-key` is non-`nil`, binding `help-form` (see Section 25.6 [Help Functions], page 604) to a non-`nil` value while calling this function causes it to evaluate `help-form` and display the result when the user presses `help-char`; it then continues to wait for a valid input character, or for keyboard-quit.

`read-multiple-choice` *prompt choices* **&optional** *help-string* [Function]
show-help long-form

Ask user a multiple choice question. *prompt* should be a string that will be displayed as the prompt.

choices is an alist where the first element in each entry is a character to be entered, the second element is a short name for the entry to be displayed while prompting (if there's room, it might be shortened), and the third, optional entry is a longer explanation that will be displayed in a help buffer if the user requests more help.

If optional argument *help-string* is non-`nil`, it should be a string with a more detailed description of all choices. It will be displayed in a help buffer instead of the default auto-generated description when the user types `?`.

If optional argument *show-help* is non-`nil`, the help buffer will be displayed immediately, before any user input. If it is a string, use it as the name of the help buffer.

If optional argument *long-form* is non-`nil`, the user will have to type in long-form answers (using `completing-read`) instead of hitting a single key. The answers must be among the second elements of the values in the *choices* list.

The return value is the matching value from *choices*.

```
(read-multiple-choice
 "Continue connecting?"
 '((?a "always" "Accept certificate for this and future sessions.")
   (?s "session only" "Accept certificate this session only.")
   (?n "no" "Refuse to use certificate, close connection.")))
```

The `read-multiple-choice-face` face is used to highlight the matching characters in the name string on graphical terminals.

22.8.3 Modifying and Translating Input Events

Emacs modifies every event it reads according to `extra-keyboard-modifiers`, then translates it through `keyboard-translate-table` (if applicable), before returning it from `read-event`.

`extra-keyboard-modifiers` [Variable]

This variable lets Lisp programs "press" the modifier keys on the keyboard. The value is a character. Only the modifiers of the character matter. Each time the user types a keyboard key, it is altered as if those modifier keys were held down. For instance, if you bind `extra-keyboard-modifiers` to `?\C-\M-a`, then all keyboard input characters typed during the scope of the binding will have the control and meta modifiers applied to them. The character `?\C-@`, equivalent to the integer 0, does not count as a control character for this purpose, but as a character with no modifiers. Thus, setting `extra-keyboard-modifiers` to zero cancels any modification.

When using a window system, the program can press any of the modifier keys in this way. Otherwise, only the `CTL` and `META` keys can be virtually pressed.

Note that this variable applies only to events that really come from the keyboard, and has no effect on mouse events or any other events.

`keyboard-translate-table` [Variable]

This terminal-local variable is the translate table for keyboard characters. It lets you reshuffle the keys on the keyboard without changing any command bindings. Its value is normally a char-table, or else `nil`. (It can also be a string or vector, but this is considered obsolete.)

If `keyboard-translate-table` is a char-table (see Section 6.6 [Char-Tables], page 119), then each character read from the keyboard is looked up in this char-table. If the value found there is non-`nil`, then it is used instead of the actual input character.

Note that this translation is the first thing that happens to a character after it is read from the terminal. Record-keeping features such as `recent-keys` and dribble files record the characters after translation.

Note also that this translation is done before the characters are supplied to input methods (see Section 34.11 [Input Methods], page 994). Use `translation-table-for-input` (see Section 34.9 [Translation of Characters], page 978), if you want to translate characters after input methods operate.

`key-translate` *from to* [Function]

This function modifies `keyboard-translate-table` to translate character code *from* into character code *to*. It creates the keyboard translate table if necessary.

Here's an example of using the `keyboard-translate-table` to make `C-x`, `C-c` and `C-v` perform the cut, copy and paste operations:

```
(key-translate "C-x" "<control-x>")
(key-translate "C-c" "<control-c>")
(key-translate "C-v" "<control-v>")
(keymap-global-set "<control-x>" 'kill-region)
```

```
(keymap-global-set "<control-c>" 'kill-ring-save)
(keymap-global-set "<control-v>" 'yank)
```

On a graphical terminal that supports extended ASCII input, you can still get the standard Emacs meanings of one of those characters by typing it with the shift key. That makes it a different character as far as keyboard translation is concerned, but it has the same usual meaning.

See Section 23.15 [Translation Keymaps], page 505, for mechanisms that translate event sequences at the level of `read-key-sequence`. If you need to translate input events that are not characters (i.e., `characterp` returns `nil` for them), you must use the event translation mechanism described there.

22.8.4 Invoking the Input Method

The event-reading functions invoke the current input method, if any (see Section 34.11 [Input Methods], page 994). If the value of `input-method-function` is non-`nil`, it should be a function; when `read-event` reads a printing character (including `SPC`) with no modifier bits, it calls that function, passing the character as an argument.

`input-method-function` [Variable]

If this is non-`nil`, its value specifies the current input method function.

Warning: don't bind this variable with `let`. It is often buffer-local, and if you bind it around reading input (which is exactly when you *would* bind it), switching buffers asynchronously while Emacs is waiting will cause the value to be restored in the wrong buffer.

The input method function should return a list of events which should be used as input. (If the list is `nil`, that means there is no input, so `read-event` waits for another event.) These events are processed before the events in `unread-command-events` (see Section 22.8.6 [Event Input Misc], page 469). Events returned by the input method function are not passed to the input method function again, even if they are printing characters with no modifier bits.

If the input method function calls `read-event` or `read-key-sequence`, it should bind `input-method-function` to `nil` first, to prevent recursion.

The input method function is not called when reading the second and subsequent events of a key sequence. Thus, these characters are not subject to input method processing. The input method function should test the values of `overriding-local-map` and `overriding-terminal-local-map`; if either of these variables is non-`nil`, the input method should put its argument into a list and return that list with no further processing.

22.8.5 Quoted Character Input

You can use the function `read-quoted-char` to ask the user to specify a character, and allow the user to specify a control or meta character conveniently, either literally or as an octal character code. The command `quoted-insert` uses this function.

`read-quoted-char` **&optional** *prompt* [Function]

This function is like `read-char`, except that if the first character read is an octal digit (0–7), it reads any number of octal digits (but stopping if a non-octal digit is

found), and returns the character represented by that numeric character code. If the character that terminates the sequence of octal digits is `RET`, it is discarded. Any other terminating character is used as input after this function returns.

Quitting is suppressed when the first character is read, so that the user can enter a *C-g*. See Section 22.11 [Quitting], page 472.

If *prompt* is supplied, it specifies a string for prompting the user. The prompt string is always displayed in the echo area, followed by a single '-'.

In the following example, the user types in the octal number 177 (which is 127 in decimal).

```
(read-quoted-char "What character")

---------- Echo Area ----------
What character 1 7 7-
---------- Echo Area ----------

     ⇒ 127
```

22.8.6 Miscellaneous Event Input Features

This section describes how to peek ahead at events without using them up, how to check for pending input, and how to discard pending input. See also the function `read-passwd` (see Section 21.9 [Reading a Password], page 418).

`unread-command-events` [Variable]

This variable holds a list of events waiting to be read as command input. The events are used in the order they appear in the list, and removed one by one as they are used.

The variable is needed because in some cases a function reads an event and then decides not to use it. Storing the event in this variable causes it to be processed normally, by the command loop or by the functions to read command input.

For example, the function that implements numeric prefix arguments reads any number of digits. When it finds a non-digit event, it must unread the event so that it can be read normally by the command loop. Likewise, incremental search uses this feature to unread events with no special meaning in a search, because these events should exit the search and then execute normally.

The reliable and easy way to extract events from a key sequence so as to put them in `unread-command-events` is to use `listify-key-sequence` (see below).

Normally you add events to the front of this list, so that the events most recently unread will be reread first.

Events read from this list are not normally added to the current command's key sequence (as returned by, e.g., `this-command-keys`), as the events will already have been added once as they were read for the first time. An element of the form `(t . event)` forces *event* to be added to the current command's key sequence.

Elements read from this list are normally recorded by the record-keeping features (see Section 42.13.2 [Recording Input], page 1291) and while defining a keyboard macro

(see Section 22.16 [Keyboard Macros], page 479). However, an element of the form `(no-record . event)` causes *event* to be processed normally without recording it.

`listify-key-sequence` *key* [Function]
This function converts the string or vector *key* to a list of individual events, which you can put in `unread-command-events`.

`input-pending-p` **&optional** *check-timers* [Function]
This function determines whether any command input is currently available to be read. It returns immediately, with value `t` if there is available input, `nil` otherwise. On rare occasions it may return `t` when no input is available.

If the optional argument *check-timers* is non-`nil`, then if no input is available, Emacs runs any timers which are ready. See Section 42.11 [Timers], page 1286.

`last-input-event` [Variable]
This variable records the last terminal input event read, whether as part of a command or explicitly by a Lisp program.

In the example below, the Lisp program reads the character `1`, ASCII code 49. It becomes the value of `last-input-event`, while `C-e` (we assume `C-x C-e` command is used to evaluate this expression) remains the value of `last-command-event`.

```
(progn (print (read-char))
       (print last-command-event)
       last-input-event)
⊣ 49
⊣ 5
⇒ 49
```

`while-no-input` *body...* [Macro]
This construct runs the *body* forms and returns the value of the last one—but only if no input arrives. If any input arrives during the execution of the *body* forms, it aborts them (working much like a quit). The `while-no-input` form returns `nil` if aborted by a real quit, and returns `t` if aborted by arrival of other input.

If a part of *body* binds `inhibit-quit` to non-`nil`, arrival of input during those parts won't cause an abort until the end of that part.

If you want to be able to distinguish all possible values computed by *body* from both kinds of abort conditions, write the code like this:

```
(while-no-input
  (list
    (progn . body)))
```

`while-no-input-ignore-events` [Variable]
This variable allow setting which special events `while-no-input` should ignore. It is a list of event symbols (see Section 22.7.13 [Event Examples], page 455).

`discard-input` [Function]
This function discards the contents of the terminal input buffer and cancels any keyboard macro that might be in the process of definition. It returns `nil`.

In the following example, the user may type a number of characters right after starting the evaluation of the form. After the `sleep-for` finishes sleeping, `discard-input` discards any characters typed during the sleep.

```
(progn (sleep-for 2)
       (discard-input))
     ⇒ nil
```

22.9 Special Events

Certain *special events* are handled at a very low level—as soon as they are read. The `read-event` function processes these events itself, and never returns them. Instead, it keeps waiting for the first event that is not special and returns that one.

Special events do not echo, they are never grouped into key sequences, and they never appear in the value of `last-command-event` or `(this-command-keys)`. They do not discard a numeric argument, they cannot be unread with `unread-command-events`, they may not appear in a keyboard macro, and they are not recorded in a keyboard macro while you are defining one.

Special events do, however, appear in `last-input-event` immediately after they are read, and this is the way for the event's definition to find the actual event.

The events types `iconify-frame`, `make-frame-visible`, `delete-frame`, `drag-n-drop`, `language-change`, and user signals like `sigusr1` are normally handled in this way. The keymap which defines how to handle special events—and which events are special—is in the variable `special-event-map` (see Section 23.9 [Controlling Active Maps], page 492).

22.10 Waiting for Elapsed Time or Input

The wait functions are designed to wait for a certain amount of time to pass or until there is input. For example, you may wish to pause in the middle of a computation to allow the user time to view the display. `sit-for` pauses and updates the screen, and returns immediately if input comes in, while `sleep-for` pauses without updating the screen.

`sit-for` *seconds* **&optional** *nodisp* [Function]

This function performs redisplay (provided there is no pending input from the user), then waits *seconds* seconds, or until input is available. The usual purpose of `sit-for` is to give the user time to read text that you display. The value is `t` if `sit-for` waited the full time with no input arriving (see Section 22.8.6 [Event Input Misc], page 469). Otherwise, the value is `nil`.

The argument *seconds* need not be an integer. If it is floating point, `sit-for` waits for a fractional number of seconds. Some systems support only a whole number of seconds; on these systems, *seconds* is rounded down.

The expression `(sit-for 0)` is equivalent to `(redisplay)`, i.e., it requests a redisplay, without any delay, if there is no pending input. See Section 41.2 [Forcing Redisplay], page 1133.

If *nodisp* is non-`nil`, then `sit-for` does not redisplay, but it still returns as soon as input is available (or when the timeout elapses).

In batch mode (see Section 42.17 [Batch Mode], page 1294), `sit-for` cannot be interrupted, even by input from the standard input descriptor. It is thus equivalent to `sleep-for`, which is described below.

It is also possible to call `sit-for` with three arguments, as `(sit-for` *`seconds millisec nodisp`*`)`, but that is considered obsolete.

`sleep-for` *seconds* **&optional** *millisec* [Function]

This function simply pauses for *seconds* seconds without updating the display. It pays no attention to available input. It returns `nil`.

The argument *seconds* need not be an integer. If it is floating point, `sleep-for` waits for a fractional number of seconds. Some systems support only a whole number of seconds; on these systems, *seconds* is rounded down.

The optional argument *millisec* specifies an additional waiting period measured in milliseconds. This adds to the period specified by *seconds*. If the system doesn't support waiting fractions of a second, you get an error if you specify nonzero *millisec*.

Use `sleep-for` when you wish to guarantee a delay.

See Section 42.5 [Time of Day], page 1275, for functions to get the current time.

22.11 Quitting

Typing `C-g` while a Lisp function is running causes Emacs to *quit* whatever it is doing. This means that control returns to the innermost active command loop.

Typing `C-g` while the command loop is waiting for keyboard input does not cause a quit; it acts as an ordinary input character. In the simplest case, you cannot tell the difference, because `C-g` normally runs the command `keyboard-quit`, whose effect is to quit. However, when `C-g` follows a prefix key, they combine to form an undefined key. The effect is to cancel the prefix key as well as any prefix argument.

In the minibuffer, `C-g` has a different definition: it aborts out of the minibuffer. This means, in effect, that it exits the minibuffer and then quits. (Simply quitting would return to the command loop *within* the minibuffer.) The reason why `C-g` does not quit directly when the command reader is reading input is so that its meaning can be redefined in the minibuffer in this way. `C-g` following a prefix key is not redefined in the minibuffer, and it has its normal effect of canceling the prefix key and prefix argument. This too would not be possible if `C-g` always quit directly.

When `C-g` does directly quit, it does so by setting the variable `quit-flag` to `t`. Emacs checks this variable at appropriate times and quits if it is not `nil`. Setting `quit-flag` non-`nil` in any way thus causes a quit.

At the level of C code, quitting cannot happen just anywhere; only at the special places that check `quit-flag`. The reason for this is that quitting at other places might leave an inconsistency in Emacs's internal state. Because quitting is delayed until a safe place, quitting cannot make Emacs crash.

Certain functions such as `read-key-sequence` or `read-quoted-char` prevent quitting entirely even though they wait for input. Instead of quitting, `C-g` serves as the requested input. In the case of `read-key-sequence`, this serves to bring about the special behavior

of `C-g` in the command loop. In the case of `read-quoted-char`, this is so that `C-q` can be used to quote a `C-g`.

You can prevent quitting for a portion of a Lisp function by binding the variable `inhibit-quit` to a non-`nil` value. Then, although `C-g` still sets `quit-flag` to `t` as usual, the usual result of this—a quit—is prevented. Eventually, `inhibit-quit` will become `nil` again, such as when its binding is unwound at the end of a `let` form. At that time, if `quit-flag` is still non-`nil`, the requested quit happens immediately. This behavior is ideal when you wish to make sure that quitting does not happen within a critical section of the program.

In some functions (such as `read-quoted-char`), `C-g` is handled in a special way that does not involve quitting. This is done by reading the input with `inhibit-quit` bound to `t`, and setting `quit-flag` to `nil` before `inhibit-quit` becomes `nil` again. This excerpt from the definition of `read-quoted-char` shows how this is done; it also shows that normal quitting is permitted after the first character of input.

```
(defun read-quoted-char (&optional prompt)
  "...documentation..."
  (let ((message-log-max nil) done (first t) (code 0) char)
    (while (not done)
      (let ((inhibit-quit first)
            ...)
        (and prompt (message "%s-" prompt))
        (setq char (read-event))
        (if inhibit-quit (setq quit-flag nil)))
      ...set the variable code...)
    code))
```

`quit-flag` [Variable]

If this variable is non-`nil`, then Emacs quits immediately, unless `inhibit-quit` is non-`nil`. Typing `C-g` ordinarily sets `quit-flag` non-`nil`, regardless of `inhibit-quit`.

`inhibit-quit` [Variable]

This variable determines whether Emacs should quit when `quit-flag` is set to a value other than `nil`. If `inhibit-quit` is non-`nil`, then `quit-flag` has no special effect.

`with-local-quit` *body...* [Macro]

This macro executes *body* forms in sequence, but allows quitting, at least locally, within *body* even if `inhibit-quit` was non-`nil` outside this construct. It returns the value of the last form in *body*, unless exited by quitting, in which case it returns `nil`.

If `inhibit-quit` is `nil` on entry to `with-local-quit`, it only executes the *body*, and setting `quit-flag` causes a normal quit. However, if `inhibit-quit` is non-`nil` so that ordinary quitting is delayed, a non-`nil` `quit-flag` triggers a special kind of local quit. This ends the execution of *body* and exits the `with-local-quit` body with `quit-flag` still non-`nil`, so that another (ordinary) quit will happen as soon as that is allowed. If `quit-flag` is already non-`nil` at the beginning of *body*, the local quit happens immediately and the body doesn't execute at all.

This macro is mainly useful in functions that can be called from timers, process filters, process sentinels, `pre-command-hook`, `post-command-hook`, and other places where `inhibit-quit` is normally bound to `t`.

`keyboard-quit` [Command]
This function signals the `quit` condition with `(signal 'quit nil)`. This is the same thing that quitting does. (See `signal` in Section 11.7.3 [Errors], page 179.)

To quit without aborting a keyboard macro definition or execution, you can signal the `minibuffer-quit` condition. This has almost the same effect as the `quit` condition except that the error handling in the command loop handles it without exiting keyboard macro definition or execution.

You can specify a character other than *C-g* to use for quitting. See the function `set-input-mode` in Section 42.13.1 [Input Modes], page 1290.

22.12 Prefix Command Arguments

Most Emacs commands can use a *prefix argument*, a number specified before the command itself. (Don't confuse prefix arguments with prefix keys.) The prefix argument is at all times represented by a value, which may be `nil`, meaning there is currently no prefix argument. Each command may use the prefix argument or ignore it.

There are two representations of the prefix argument: *raw* and *numeric*. The editor command loop uses the raw representation internally, and so do the Lisp variables that store the information, but commands can request either representation.

Here are the possible values of a raw prefix argument:

- `nil`, meaning there is no prefix argument. Its numeric value is 1, but numerous commands make a distinction between `nil` and the integer 1.
- An integer, which stands for itself.
- A list of one element, which is an integer. This form of prefix argument results from one or a succession of *C-u*s with no digits. The numeric value is the integer in the list, but some commands make a distinction between such a list and an integer alone.
- The symbol `-`. This indicates that *M--* or *C-u* `-` was typed, without following digits. The equivalent numeric value is −1, but some commands make a distinction between the integer −1 and the symbol `-`.

We illustrate these possibilities by calling the following function with various prefixes:

```
(defun display-prefix (arg)
  "Display the value of the raw prefix arg."
  (interactive "P")
  (message "%s" arg))
```

Here are the results of calling `display-prefix` with various raw prefix arguments:

```
          M-x display-prefix  ⊣ nil

C-u       M-x display-prefix  ⊣ (4)

C-u C-u M-x display-prefix  ⊣ (16)
```

```
C-u 3   M-x display-prefix  ⊣ 3

M-3     M-x display-prefix  ⊣ 3       ; (Same as C-u 3.)

C-u -   M-x display-prefix  ⊣ -

M--     M-x display-prefix  ⊣ -       ; (Same as C-u -.)

C-u - 7 M-x display-prefix  ⊣ -7

M-- 7   M-x display-prefix  ⊣ -7      ; (Same as C-u -7.)
```

Emacs uses two variables to store the prefix argument: `prefix-arg` and `current-prefix-arg`. Commands such as `universal-argument` that set up prefix arguments for other commands store them in `prefix-arg`. In contrast, `current-prefix-arg` conveys the prefix argument to the current command, so setting it has no effect on the prefix arguments for future commands.

Normally, commands specify which representation to use for the prefix argument, either numeric or raw, in the `interactive` specification. (See Section 22.2.1 [Using Interactive], page 425.) Alternatively, functions may look at the value of the prefix argument directly in the variable `current-prefix-arg`, but this is less clean.

`prefix-numeric-value` *arg* [Function]
: This function returns the numeric meaning of a valid raw prefix argument value, *arg*. The argument may be a symbol, a number, or a list. If it is `nil`, the value 1 is returned; if it is `-`, the value −1 is returned; if it is a number, that number is returned; if it is a list, the CAR of that list (which should be a number) is returned.

`current-prefix-arg` [Variable]
: This variable holds the raw prefix argument for the *current* command. Commands may examine it directly, but the usual method for accessing it is with `(interactive "P")`.

`prefix-arg` [Variable]
: The value of this variable is the raw prefix argument for the *next* editing command. Commands such as `universal-argument` that specify prefix arguments for the following command work by setting this variable.

`last-prefix-arg` [Variable]
: The raw prefix argument value used by the previous command.

The following commands exist to set up prefix arguments for the following command. Do not call them for any other reason.

`universal-argument` [Command]
: This command reads input and specifies a prefix argument for the following command. Don't call this command yourself unless you know what you are doing.

`digit-argument` *arg* [Command]
: This command adds to the prefix argument for the following command. The argument *arg* is the raw prefix argument as it was before this command; it is used to compute the updated prefix argument. Don't call this command yourself unless you know what you are doing.

`negative-argument` *arg* [Command]
: This command adds to the numeric argument for the next command. The argument *arg* is the raw prefix argument as it was before this command; its value is negated to form the new prefix argument. Don't call this command yourself unless you know what you are doing.

22.13 Recursive Editing

The Emacs command loop is entered automatically when Emacs starts up. This top-level invocation of the command loop never exits; it keeps running as long as Emacs does. Lisp programs can also invoke the command loop. Since this makes more than one activation of the command loop, we call it *recursive editing*. A recursive editing level has the effect of suspending whatever command invoked it and permitting the user to do arbitrary editing before resuming that command.

The commands available during recursive editing are the same ones available in the top-level editing loop and defined in the keymaps. Only a few special commands exit the recursive editing level; the others return to the recursive editing level when they finish. (The special commands for exiting are always available, but they do nothing when recursive editing is not in progress.)

All command loops, including recursive ones, set up all-purpose error handlers so that an error in a command run from the command loop will not exit the loop.

Minibuffer input is a special kind of recursive editing. It has a few special wrinkles, such as enabling display of the minibuffer and the minibuffer window, but fewer than you might suppose. Certain keys behave differently in the minibuffer, but that is only because of the minibuffer's local map; if you switch windows, you get the usual Emacs commands.

To invoke a recursive editing level, call the function `recursive-edit`. This function contains the command loop; it also contains a call to `catch` with tag `exit`, which makes it possible to exit the recursive editing level by throwing to `exit` (see Section 11.7.1 [Catch and Throw], page 177). Throwing a `t` value causes `recursive-edit` to quit, so that control returns to the command loop one level up. This is called *aborting*, and is done by `C-]` (`abort-recursive-edit`). Similarly, you can throw a string value to make `recursive-edit` signal an error, printing this string as the message. If you throw a function, `recursive-edit` will call it without arguments before returning. Throwing any other value, will make `recursive-edit` return normally to the function that called it. The command `C-M-c` (`exit-recursive-edit`) does this.

Most applications should not use recursive editing, except as part of using the minibuffer. Usually it is more convenient for the user if you change the major mode of the current buffer temporarily to a special major mode, which should have a command to go back to the previous mode. (The `e` command in Rmail uses this technique.) Or, if you wish to give the user different text to edit recursively, create and select a new buffer in a special mode.

In this mode, define a command to complete the processing and go back to the previous buffer. (The `m` command in Rmail does this.)

Recursive edits are useful in debugging. You can insert a call to `debug` into a function definition as a sort of breakpoint, so that you can look around when the function gets there. `debug` invokes a recursive edit but also provides the other features of the debugger.

Recursive editing levels are also used when you type `C-r` in `query-replace` or use `C-x q` (`kbd-macro-query`).

`recursive-edit` [Command]

This function invokes the editor command loop. It is called automatically by the initialization of Emacs, to let the user begin editing. When called from a Lisp program, it enters a recursive editing level.

If the current buffer is not the same as the selected window's buffer, `recursive-edit` saves and restores the current buffer. Otherwise, if you switch buffers, the buffer you switched to is current after `recursive-edit` returns.

In the following example, the function `simple-rec` first advances point one word, then enters a recursive edit, printing out a message in the echo area. The user can then do any editing desired, and then type `C-M-c` to exit and continue executing `simple-rec`.

```
(defun simple-rec ()
  (forward-word 1)
  (message "Recursive edit in progress")
  (recursive-edit)
  (forward-word 1))
     ⇒ simple-rec
(simple-rec)
     ⇒ nil
```

`exit-recursive-edit` [Command]

This function exits from the innermost recursive edit (including minibuffer input). Its definition is effectively `(throw 'exit nil)`.

`abort-recursive-edit` [Command]

This function aborts the command that requested the innermost recursive edit (including minibuffer input), by signaling `quit` after exiting the recursive edit. Its definition is effectively `(throw 'exit t)`. See Section 22.11 [Quitting], page 472.

`top-level` [Command]

This function exits all recursive editing levels; it does not return a value, as it jumps completely out of any computation directly back to the main command loop.

`recursion-depth` [Function]

This function returns the current depth of recursive edits. When no recursive edit is active, it returns 0.

22.14 Disabling Commands

Disabling a command marks the command as requiring user confirmation before it can be executed. Disabling is used for commands which might be confusing to beginning users, to prevent them from using the commands by accident.

The low-level mechanism for disabling a command is to put a non-`nil` `disabled` property on the Lisp symbol for the command. These properties are normally set up by the user's init file (see Section 42.1.2 [Init File], page 1263) with Lisp expressions such as this:

```
(put 'upcase-region 'disabled t)
```

For a few commands, these properties are present by default (you can remove them in your init file if you wish).

If the value of the `disabled` property is a string, the message saying the command is disabled includes that string. For example:

```
(put 'delete-region 'disabled
     "Text deleted this way cannot be yanked back!\n")
```

See Section "Disabling" in *The GNU Emacs Manual*, for the details on what happens when a disabled command is invoked interactively. Disabling a command has no effect on calling it as a function from Lisp programs.

The value of the `disabled` property can also be a list where the first element is the symbol `query`. In that case, the user will be queried whether to execute the command. The second element in the list should be `nil` or non-`nil` to say whether to use `y-or-n-p` or `yes-or-no-p`, respectively, and the third element is the question to use. The `command-query` convenience function should be used to enable querying for a command.

`enable-command` *command* [Command]
: Allow *command* (a symbol) to be executed without special confirmation from now on, and alter the user's init file (see Section 42.1.2 [Init File], page 1263) so that this will apply to future sessions.

`disable-command` *command* [Command]
: Require special confirmation to execute *command* from now on, and alter the user's init file so that this will apply to future sessions.

`disabled-command-function` [Variable]
: The value of this variable should be a function. When the user invokes a disabled command interactively, this function is called instead of the disabled command. It can use `this-command-keys` to determine what the user typed to run the command, and thus find the command itself.

  The value may also be `nil`. Then all commands work normally, even disabled ones.

  By default, the value is a function that asks the user whether to proceed.

22.15 Command History

The command loop keeps a history of the complex commands that have been executed, to make it convenient to repeat these commands. A *complex command* is one for which the interactive argument reading uses the minibuffer. This includes any `M-x` command, any `M-:` command, and any command whose `interactive` specification reads an argument from the

minibuffer. Explicit use of the minibuffer during the execution of the command itself does not cause the command to be considered complex.

`command-history` [Variable]
: This variable's value is a list of recent complex commands, each represented as a form to evaluate. It continues to accumulate all complex commands for the duration of the editing session, but when it reaches the maximum size (see Section 21.4 [Minibuffer History], page 393), the oldest elements are deleted as new ones are added.

```
command-history
⇒ ((switch-to-buffer "chistory.texi")
    (describe-key "^X^[")
    (visit-tags-table "~/emacs/src/")
    (find-tag "repeat-complex-command"))
```

This history list is actually a special case of minibuffer history (see Section 21.4 [Minibuffer History], page 393), with one special twist: the elements are expressions rather than strings.

There are a number of commands devoted to the editing and recall of previous commands. The commands `repeat-complex-command`, and `list-command-history` are described in the user manual (see Section "Repetition" in *The GNU Emacs Manual*). Within the minibuffer, the usual minibuffer history commands are available.

22.16 Keyboard Macros

A *keyboard macro* is a canned sequence of input events that can be considered a command and made the definition of a key. The Lisp representation of a keyboard macro is a string or vector containing the events. Don't confuse keyboard macros with Lisp macros (see Chapter 14 [Macros], page 269).

`execute-kbd-macro` *kbdmacro* **&optional** *count loopfunc* [Function]
: This function executes *kbdmacro* as a sequence of events. If *kbdmacro* is a string or vector, then the events in it are executed exactly as if they had been input by the user. The sequence is *not* expected to be a single key sequence; normally a keyboard macro definition consists of several key sequences concatenated.

  If *kbdmacro* is a symbol, then its function definition is used in place of *kbdmacro*. If that is another symbol, this process repeats. Eventually the result should be a string or vector. If the result is not a symbol, string, or vector, an error is signaled.

  The argument *count* is a repeat count; *kbdmacro* is executed that many times. If *count* is omitted or `nil`, *kbdmacro* is executed once. If it is 0, *kbdmacro* is executed over and over until it encounters an error or a failing search.

  If *loopfunc* is non-`nil`, it is a function that is called, without arguments, prior to each iteration of the macro. If *loopfunc* returns `nil`, then this stops execution of the macro.

  See Section 22.8.2 [Reading One Event], page 463, for an example of using `execute-kbd-macro`.

`executing-kbd-macro` [Variable]

This variable contains the string or vector that defines the keyboard macro that is currently executing. It is `nil` if no macro is currently executing. A command can test this variable so as to behave differently when run from an executing macro. Do not set this variable yourself.

`defining-kbd-macro` [Variable]

This variable is non-`nil` if and only if a keyboard macro is being defined. A command can test this variable so as to behave differently while a macro is being defined. The value is `append` while appending to the definition of an existing macro. The commands `start-kbd-macro`, `kmacro-start-macro` and `end-kbd-macro` set this variable—do not set it yourself.

The variable is always local to the current terminal and cannot be buffer-local. See Section 30.2 [Multiple Terminals], page 789.

`last-kbd-macro` [Variable]

This variable is the definition of the most recently defined keyboard macro. Its value is a string or vector, or `nil`.

The variable is always local to the current terminal and cannot be buffer-local. See Section 30.2 [Multiple Terminals], page 789.

`kbd-macro-termination-hook` [Variable]

This normal hook is run when a keyboard macro terminates, regardless of what caused it to terminate (reaching the macro end or an error which ended the macro prematurely).

23 Keymaps

The command bindings of input events are recorded in data structures called *keymaps*. Each entry in a keymap associates (or *binds*) an individual event type, either to another keymap or to a command. When an event type is bound to a keymap, that keymap is used to look up the next input event; this continues until a command is found. The whole process is called *key lookup*.

23.1 Key Sequences

A *key sequence*, or *key* for short, is a sequence of one or more input events that form a unit. Input events include characters, function keys, mouse actions, or system events external to Emacs, such as `iconify-frame` (see Section 22.7 [Input Events], page 441). The Emacs Lisp representation for a key sequence is a string or vector. Unless otherwise stated, any Emacs Lisp function that accepts a key sequence as an argument can handle both representations.

In the string representation, alphanumeric characters ordinarily stand for themselves; for example, `"a"` represents `a` and `"2"` represents *2*. Control character events are prefixed by the substring `"\C-"`, and meta characters by `"\M-"`; for example, `"\C-x"` represents the key *C-x*. In addition, the `TAB`, `RET`, `ESC`, and `DEL` events are represented by `"\t"`, `"\r"`, `"\e"`, and `"\d"` respectively. The string representation of a complete key sequence is the concatenation of the string representations of the constituent events; thus, `"\C-xl"` represents the key sequence *C-x l*.

Key sequences containing function keys, mouse button events, system events, or non-ASCII characters such as *C-=* or *H-a* cannot be represented as strings; they have to be represented as vectors.

In the vector representation, each element of the vector represents an input event, in its Lisp form. See Section 22.7 [Input Events], page 441. For example, the vector `[?\C-x ?l]` represents the key sequence *C-x l*.

For examples of key sequences written in string and vector representations, Section "Init Rebinding" in *The GNU Emacs Manual*.

`kbd` *keyseq-text* [Function]

This function converts the text *keyseq-text* (a string constant) into a key sequence (a string or vector constant). The contents of *keyseq-text* should use the same syntax as in the buffer invoked by the *C-x C-k* `RET` (`kmacro-edit-macro`) command; in particular, you must surround function key names with '`<...>`'. See Section "Edit Keyboard Macro" in *The GNU Emacs Manual*.

```
(kbd "C-x") ⇒ "\C-x"
(kbd "C-x C-f") ⇒ "\C-x\C-f"
(kbd "C-x 4 C-f") ⇒ "\C-x4\C-f"
(kbd "X") ⇒ "X"
(kbd "RET") ⇒ "^M"
(kbd "C-c SPC") ⇒ "\C-c "
(kbd "<f1> SPC") ⇒ [f1 32]
(kbd "C-M-<down>") ⇒ [C-M-down]
```

The `kbd` function is very permissive, and will try to return something sensible even if the syntax used isn't completely conforming. To check whether the syntax is actually valid, use the `key-valid-p` function.

23.2 Keymap Basics

A keymap is a Lisp data structure that specifies *key bindings* for various key sequences.

A single keymap directly specifies definitions for individual events. When a key sequence consists of a single event, its binding in a keymap is the keymap's definition for that event. The binding of a longer key sequence is found by an iterative process: first find the definition of the first event (which must itself be a keymap); then find the second event's definition in that keymap, and so on until all the events in the key sequence have been processed.

If the binding of a key sequence is a keymap, we call the key sequence a *prefix key*. Otherwise, we call it a *complete key* (because no more events can be added to it). If the binding is `nil`, we call the key *undefined*. Examples of prefix keys are `C-c`, `C-x`, and `C-x 4`. Examples of defined complete keys are `X`, `RET`, and `C-x 4 C-f`. Examples of undefined complete keys are `C-x C-g`, and `C-c 3`. See Section 23.6 [Prefix Keys], page 489, for more details.

The rule for finding the binding of a key sequence assumes that the intermediate bindings (found for the events before the last) are all keymaps; if this is not so, the sequence of events does not form a unit—it is not really one key sequence. In other words, removing one or more events from the end of any valid key sequence must always yield a prefix key. For example, `C-f C-n` is not a key sequence; `C-f` is not a prefix key, so a longer sequence starting with `C-f` cannot be a key sequence.

The set of possible multi-event key sequences depends on the bindings for prefix keys; therefore, it can be different for different keymaps, and can change when bindings are changed. However, a one-event sequence is always a key sequence, because it does not depend on any prefix keys for its well-formedness.

At any time, several primary keymaps are *active*—that is, in use for finding key bindings. These are the *global map*, which is shared by all buffers; the *local keymap*, which is usually associated with a specific major mode; and zero or more *minor mode keymaps*, which belong to currently enabled minor modes. (Not all minor modes have keymaps.) The local keymap bindings shadow (i.e., take precedence over) the corresponding global bindings. The minor mode keymaps shadow both local and global keymaps. See Section 23.7 [Active Keymaps], page 490, for details.

23.3 Format of Keymaps

Each keymap is a list whose CAR is the symbol `keymap`. The remaining elements of the list define the key bindings of the keymap. A symbol whose function definition is a keymap is also a keymap. Use the function `keymapp` (see below) to test whether an object is a keymap.

Several kinds of elements may appear in a keymap, after the symbol `keymap` that begins it:

`(type . binding)`
: This specifies one binding, for events of type *type*. Each ordinary binding applies to events of a particular *event type*, which is always a character or a

symbol. See Section 22.7.14 [Classifying Events], page 456. In this kind of binding, *binding* is a command.

`(`*type item-name* `.` *binding*`)`
: This specifies a binding which is also a simple menu item that displays as *item-name* in the menu. See Section 23.18.1.1 [Simple Menu Items], page 511.

`(`*type item-name help-string* `.` *binding*`)`
: This is a simple menu item with help string *help-string*.

`(`*type* `menu-item` `.` *details*`)`
: This specifies a binding which is also an extended menu item. This allows use of other features. See Section 23.18.1.2 [Extended Menu Items], page 512.

`(t .` *binding*`)`
: This specifies a *default key binding*; any event not bound by other elements of the keymap is given *binding* as its binding. Default bindings allow a keymap to bind all possible event types without having to enumerate all of them. A keymap that has a default binding completely masks any lower-precedence keymap, except for events explicitly bound to `nil` (see below).

char-table
: If an element of a keymap is a char-table, it counts as holding bindings for all character events with no modifier bits (see [modifier bits], page 14): the element whose index is *c* is the binding for the character *c*. This is a compact way to record lots of bindings. A keymap with such a char-table is called a *full keymap*. Other keymaps are called *sparse keymaps*.

vector
: This kind of element is similar to a char-table: the element whose index is *c* is the binding for the character *c*. Since the range of characters that can be bound this way is limited by the vector size, and vector creation allocates space for all character codes from 0 up, this format should not be used except for creating menu keymaps (see Section 23.18 [Menu Keymaps], page 511), where the bindings themselves don't matter.

string
: Aside from elements that specify bindings for keys, a keymap can also have a string as an element. This is called the *overall prompt string* and makes it possible to use the keymap as a menu. See Section 23.18.1 [Defining Menus], page 511.

`(keymap ...)`
: If an element of a keymap is itself a keymap, it counts as if this inner keymap were inlined in the outer keymap. This is used for multiple-inheritance, such as in `make-composed-keymap`.

When the binding is `nil`, it doesn't constitute a definition but it does take precedence over a default binding or a binding in the parent keymap. On the other hand, a binding of `nil` does *not* override lower-precedence keymaps; thus, if the local map gives a binding of `nil`, Emacs uses the binding from the global map.

Keymaps do not directly record bindings for the meta characters. Instead, meta characters are regarded for purposes of key lookup as sequences of two characters, the first of which is `ESC` (or whatever is currently the value of `meta-prefix-char`). Thus, the key

M-a is internally represented as ESC a, and its global binding is found at the slot for a in esc-map (see Section 23.6 [Prefix Keys], page 489).

This conversion applies only to characters, not to function keys or other input events; thus, *M*-end has nothing to do with ESC end.

Here as an example is the local keymap for Lisp mode, a sparse keymap. It defines bindings for DEL, *C-c C-z*, *C-M-q*, and *C-M-x* (the actual value also contains a menu binding, which is omitted here for the sake of brevity).

```
lisp-mode-map
⇒
(keymap
 (3 keymap
    ;; C-c C-z
    (26 . run-lisp))
 (27 keymap
     ;; C-M-x, treated as ESC C-x
     (24 . lisp-send-defun))
 ;; This part is inherited from lisp-mode-shared-map.
 keymap
 ;; DEL
 (127 . backward-delete-char-untabify)
 (27 keymap
     ;; C-M-q, treated as ESC C-q
     (17 . indent-sexp)))
```

keymapp *object* [Function]

This function returns t if *object* is a keymap, nil otherwise. More precisely, this function tests for a list whose CAR is keymap, or for a symbol whose function definition satisfies keymapp.

```
(keymapp '(keymap))
    ⇒ t
(fset 'foo '(keymap))
(keymapp 'foo)
    ⇒ t
(keymapp (current-global-map))
    ⇒ t
```

23.4 Creating Keymaps

Here we describe the functions for creating keymaps.

make-sparse-keymap **&optional** *prompt* [Function]

This function creates and returns a new sparse keymap with no entries. (A sparse keymap is the kind of keymap you usually want.) The new keymap does not contain a char-table, unlike make-keymap, and does not bind any events.

```
(make-sparse-keymap)
    ⇒ (keymap)
```

If you specify *prompt*, that becomes the overall prompt string for the keymap. You should specify this only for menu keymaps (see Section 23.18.1 [Defining Menus], page 511). A keymap with an overall prompt string will always present a mouse menu or a keyboard menu if it is active for looking up the next input event. Don't specify an overall prompt string for the main map of a major or minor mode, because that would cause the command loop to present a keyboard menu every time.

`make-keymap` **&optional** *prompt* [Function]

This function creates and returns a new full keymap. That keymap contains a char-table (see Section 6.6 [Char-Tables], page 119) with slots for all characters without modifiers. The new keymap initially binds all these characters to `nil`, and does not bind any other kind of event. The argument *prompt* specifies a prompt string, as in `make-sparse-keymap`.

```
(make-keymap)
    ⇒ (keymap #^[nil nil keymap nil nil nil ...])
```

A full keymap is more efficient than a sparse keymap when it holds lots of bindings; for just a few, the sparse keymap is better.

`define-keymap` **&key** *options...* **&rest** *pairs...* [Function]

You can create a keymap with the functions described above, and then use `keymap-set` (see Section 23.12 [Changing Key Bindings], page 498) to specify key bindings in that map. When writing modes, however, you frequently have to bind a large number of keys at once, and using `keymap-set` on them all can be tedious and error-prone. Instead you can use `define-keymap`, which creates a keymap and binds a number of keys. Here's a very basic example:

```
(define-keymap
  "n" #'forward-line
  "f" #'previous-line
  "C-c C-c" #'quit-window)
```

This function creates a new sparse keymap, defines the keystrokes in *pairs*, and returns the new keymap. It signals an error if there are duplicate key bindings in *pairs*.

pairs is a list of alternating key bindings and key definitions, as accepted by `keymap-set`. In addition, the key can be the special symbol `:menu`, in which case the definition should be a menu definition as accepted by `easy-menu-define` (see Section 23.18.8 [Easy Menu], page 522). Here's a brief example of this usage:

```
(define-keymap :full t
  "g" #'eww-reload
  :menu '("Eww"
          ["Exit" quit-window t]
          ["Reload" eww-reload t]))
```

A number of keywords can be used before the key/definition pairs to change features of the new keymap. If any of the feature keywords is missing from the `define-keymap` call, the default value for that feature is `nil`. Here's a list of the available feature keywords:

`:full` If non-`nil`, create a char-table keymap (as from `make-keymap`) instead of a sparse keymap (as from `make-sparse-keymap` (see Section 23.4 [Creating Keymaps], page 484). A sparse keymap is the default.

`:parent` If non-`nil`, the value should be a keymap to use as the parent (see Section 23.5 [Inheritance and Keymaps], page 488).

`:keymap` If non-`nil`, the value should be a keymap. Instead of creating a new keymap, the specified keymap is modified instead.

`:suppress`
If non-`nil`, the keymap will be suppressed with `suppress-keymap` (see Section 23.12 [Changing Key Bindings], page 498). By default, digits and the minus sign are exempt from suppressing, but if the value is `nodigits`, this suppresses digits and minus-sign like it does with other characters.

`:name` If non-`nil`, the value should be a string to use as the menu for the keymap if you use it as a menu with `x-popup-menu` (see Section 30.17 [Pop-Up Menus], page 839).

`:prefix` If non-`nil`, the value should be a symbol to be used as a prefix command (see Section 23.6 [Prefix Keys], page 489). If this is the case, this symbol is returned by `define-keymap` instead of the map itself.

`defvar-keymap` *name* **&key** *options...* **&rest** *pairs...* [Macro]

By far, the most common thing to do with a keymap is to bind it to a variable. This is what virtually all modes do—a mode called `foo` almost always has a variable called `foo-mode-map`.

This macro defines *name* as a variable, passes *options* and *pairs* to `define-keymap`, and uses the result as the default value for the variable. It signals an error if there are duplicate key bindings in *pairs*.

options is like the keywords in `define-keymap`, but there's an additional `:doc` keyword that provides the doc string for the defined variable.

Here's an example:

```
(defvar-keymap eww-textarea-map
  :parent text-mode-map
  :doc "Keymap for the eww text area."
  "RET" #'forward-line
  "TAB" #'shr-next-link)
```

Each command in the keymap can be marked as 'repeatable', i.e. usable in `repeat-mode`, by putting a `repeat-map` property on it, e.g.

```
(put 'undo 'repeat-map 'undo-repeat-map)
```

where the value of the property is the map to be used by `repeat-mode`.

To avoid repetitive invocations of `put`, `defvar-keymap` also has a `:repeat` keyword, which can be used to specify which commands in the keymap are usable by `repeat-mode`. The following values are available:

`t` This means all the commands in the keymap are repeatable, and is the most common usage.

`(:enter (commands ...) :exit (commands ...))`

This specifies that the commands in the `:enter` list enter `repeat-mode`, and the commands in the `:exit` list exit repeat mode.

If the `:enter` list is empty, then all commands in the map enter `repeat-mode`. Specifying one or more commands in this list is useful if there is a command which doesn't exist in the map being defined, but which should have the `repeat-map` property.

If the `:exit` list is empty then no commands in the map exit `repeat-mode`. Specifying one or more commands in this list is useful if the keymap being defined contains a command that should not have the `repeat-map` property.

In order to make e.g. `u` repeat the `undo` command, the following two stanzas are equivalent:

```
(defvar-keymap undo-repeat-map
  "u" #'undo)
(put 'undo 'repeat-map 'undo-repeat-map)
```

and

```
(defvar-keymap undo-repeat-map
  :repeat t
  "u" #'undo)
```

The latter is preferred when there are many commands in the map, all of which should be repeatable.

`copy-keymap` *keymap* [Function]

This function returns a copy of *keymap*. This is almost never needed. If you want a keymap that's like another yet with a few changes, you should use map inheritance rather than copying. I.e., something like:

```
(let ((map (make-sparse-keymap)))
  (set-keymap-parent map <theirmap>)
  (keymap-set map ...)
  ...)
```

When performing `copy-keymap`, any keymaps that appear directly as bindings in *keymap* are also copied recursively, and so on to any number of levels. However, recursive copying does not take place when the definition of a character is a symbol whose function definition is a keymap; the same symbol appears in the new copy.

```
(setq map (copy-keymap (current-local-map)))
⇒ (keymap
     ;; (This implements meta characters.)
     (27 keymap
         (83 . center-paragraph)
         (115 . center-line))
     (9 . tab-to-tab-stop))

(eq map (current-local-map))
    ⇒ nil
```

```
(equal map (current-local-map))
    ⇒ t
```

23.5 Inheritance and Keymaps

A keymap can inherit the bindings of another keymap, which we call the *parent keymap*. Such a keymap looks like this:

```
(keymap elements... . parent-keymap)
```

The effect is that this keymap inherits all the bindings of *parent-keymap*, whatever they may be at the time a key is looked up, but can add to them or override them with *elements*.

If you change the bindings in *parent-keymap* using `keymap-set` or other key-binding functions, these changed bindings are visible in the inheriting keymap, unless shadowed by the bindings made by *elements*. The converse is not true: if you use `keymap-set` to change bindings in the inheriting keymap, these changes are recorded in *elements*, but have no effect on *parent-keymap*.

The proper way to construct a keymap with a parent is to use `set-keymap-parent`; if you have code that directly constructs a keymap with a parent, please convert the program to use `set-keymap-parent` instead.

`keymap-parent` *keymap* [Function]
: This returns the parent keymap of *keymap*. If *keymap* has no parent, `keymap-parent` returns `nil`.

`set-keymap-parent` *keymap parent* [Function]
: This sets the parent keymap of *keymap* to *parent*, and returns *parent*. If *parent* is `nil`, this function gives *keymap* no parent at all.

  If *keymap* has submaps (bindings for prefix keys), they too receive new parent keymaps that reflect what *parent* specifies for those prefix keys.

Here is an example showing how to make a keymap that inherits from `text-mode-map`:

```
(let ((map (make-sparse-keymap)))
  (set-keymap-parent map text-mode-map)
  map)
```

A non-sparse keymap can have a parent too, but this is not very useful. A non-sparse keymap always specifies something as the binding for every numeric character code without modifier bits, even if it is `nil`, so these character's bindings are never inherited from the parent keymap.

Sometimes you want to make a keymap that inherits from more than one map. You can use the function `make-composed-keymap` for this.

`make-composed-keymap` *maps* **&optional** *parent* [Function]
: This function returns a new keymap composed of the existing keymap(s) *maps*, and optionally inheriting from a parent keymap *parent*. *maps* can be a single keymap or a list of more than one. When looking up a key in the resulting new map, Emacs searches in each of the *maps* in turn, and then in *parent*, stopping at the first match. A `nil` binding in any one of *maps* overrides any binding in *parent*, but it does not override any non-`nil` binding in any other of the *maps*.

For example, here is how Emacs sets the parent of `help-mode-map`, such that it inherits from both `button-buffer-map` and `special-mode-map`:

```
(defvar-keymap help-mode-map
  :parent (make-composed-keymap button-buffer-map
                                special-mode-map)
  ...)
```

23.6 Prefix Keys

A *prefix key* is a key sequence whose binding is a keymap. The keymap defines what to do with key sequences that extend the prefix key. For example, `C-x` is a prefix key, and it uses a keymap that is also stored in the variable `ctl-x-map`. This keymap defines bindings for key sequences starting with `C-x`.

Some of the standard Emacs prefix keys use keymaps that are also found in Lisp variables:

- `esc-map` is the global keymap for the `ESC` prefix key. Thus, the global definitions of all meta characters are actually found here. This map is also the function definition of `ESC-prefix`.
- `help-map` is the global keymap for the `C-h` prefix key.
- `mode-specific-map` is the global keymap for the prefix key `C-c`. This map is actually global, not mode-specific, but its name provides useful information about `C-c` in the output of `C-h b` (`display-bindings`), since the main use of this prefix key is for mode-specific bindings.
- `ctl-x-map` is the global keymap used for the `C-x` prefix key. This map is found via the function cell of the symbol `Control-X-prefix`.
- `mule-keymap` is the global keymap used for the `C-x RET` prefix key.
- `ctl-x-4-map` is the global keymap used for the `C-x 4` prefix key.
- `ctl-x-5-map` is the global keymap used for the `C-x 5` prefix key.
- `2C-mode-map` is the global keymap used for the `C-x 6` prefix key.
- `tab-prefix-map` is the global keymap used for the `C-x t` prefix key.
- `vc-prefix-map` is the global keymap used for the `C-x v` prefix key.
- `goto-map` is the global keymap used for the `M-g` prefix key.
- `search-map` is the global keymap used for the `M-s` prefix key.
- The other Emacs prefix keys are `C-x @`, `C-x a i`, `C-x ESC` and `ESC ESC`. They use keymaps that have no special names.

The keymap binding of a prefix key is used for looking up the event that follows the prefix key. (It may instead be a symbol whose function definition is a keymap. The effect is the same, but the symbol serves as a name for the prefix key.) Thus, the binding of `C-x` is the symbol `Control-X-prefix`, whose function cell holds the keymap for `C-x` commands. (The same keymap is also the value of `ctl-x-map`.)

Prefix key definitions can appear in any active keymap. The definitions of `C-c`, `C-x`, `C-h` and `ESC` as prefix keys appear in the global map, so these prefix keys are always available. Major and minor modes can redefine a key as a prefix by putting a prefix key definition for it in the local map or the minor mode's map. See Section 23.7 [Active Keymaps], page 490.

If a key is defined as a prefix in more than one active map, then its various definitions are in effect merged: the commands defined in the minor mode keymaps come first, followed by those in the local map's prefix definition, and then by those from the global map.

In the following example, we make `C-p` a prefix key in the local keymap, in such a way that `C-p` is identical to `C-x`. Then the binding for `C-p C-f` is the function `find-file`, just like `C-x C-f`. By contrast, the key sequence `C-p 9` is not found in any active keymap.

```
(use-local-map (make-sparse-keymap))
    ⇒ nil
(keymap-local-set "C-p" ctl-x-map)
    ⇒ (keymap #^[nil nil keymap ...
(keymap-lookup nil "C-p C-f")
    ⇒ find-file
(keymap-lookup nil "C-p 9")
    ⇒ nil
```

`define-prefix-command` *symbol* **&optional** *mapvar prompt* [Function]

This function prepares *symbol* for use as a prefix key's binding: it creates a sparse keymap and stores it as *symbol*'s function definition. Subsequently binding a key sequence to *symbol* will make that key sequence into a prefix key. The return value is `symbol`.

This function also sets *symbol* as a variable, with the keymap as its value. But if *mapvar* is non-`nil`, it sets *mapvar* as a variable instead.

If *prompt* is non-`nil`, that becomes the overall prompt string for the keymap. The prompt string should be given for menu keymaps (see Section 23.18.1 [Defining Menus], page 511).

23.7 Active Keymaps

Emacs contains many keymaps, but at any time only a few keymaps are *active*. When Emacs receives user input, it translates the input event (see Section 23.15 [Translation Keymaps], page 505), and looks for a key binding in the active keymaps.

Usually, the active keymaps are: (i) the keymap specified by the `keymap` property, (ii) the keymaps of enabled minor modes, (iii) the current buffer's local keymap, and (iv) the global keymap, in that order. Emacs searches for each input key sequence in all these keymaps.

Of these usual keymaps, the highest-precedence one is specified by the `keymap` text or overlay property at point, if any. (For a mouse input event, Emacs uses the event position instead of point; see the next section for details.)

Next in precedence are keymaps specified by enabled minor modes. These keymaps, if any, are specified by the variables `emulation-mode-map-alists`, `minor-mode-overriding-map-alist`, and `minor-mode-map-alist`. See Section 23.9 [Controlling Active Maps], page 492.

Next in precedence is the buffer's *local keymap*, containing key bindings specific to the buffer. The minibuffer also has a local keymap (see Section 21.1 [Intro to Minibuffers], page 386). If there is a `local-map` text or overlay property at point, that specifies the local keymap to use, in place of the buffer's default local keymap.

The local keymap is normally set by the buffer's major mode, and every buffer with the same major mode shares the same local keymap. Hence, if you call `keymap-local-set` (see Section 23.16 [Key Binding Commands], page 507) to change the local keymap in one buffer, that also affects the local keymaps in other buffers with the same major mode.

Finally, the *global keymap* contains key bindings that are defined regardless of the current buffer, such as `C-f`. It is always active, and is bound to the variable `global-map`.

Apart from the above usual keymaps, Emacs provides special ways for programs to make other keymaps active. Firstly, the variable `overriding-local-map` specifies a keymap that replaces the usual active keymaps, except for the global keymap. Secondly, the terminal-local variable `overriding-terminal-local-map` specifies a keymap that takes precedence over *all* other keymaps (including `overriding-local-map`); this is normally used for modal/transient key bindings (the function `set-transient-map` provides a convenient interface for this). See Section 23.9 [Controlling Active Maps], page 492, for details.

Making keymaps active is not the only way to use them. Keymaps are also used in other ways, such as for translating events within `read-key-sequence`. See Section 23.15 [Translation Keymaps], page 505.

See Appendix G [Standard Keymaps], page 1400, for a list of some standard keymaps.

`current-active-maps` **&optional** *olp position* [Function]
: This returns the list of active keymaps that would be used by the command loop in the current circumstances to look up a key sequence. Normally it ignores `overriding-local-map` and `overriding-terminal-local-map`, but if *olp* is non-`nil` then it pays attention to them. *position* can optionally be either an event position as returned by `event-start` or a buffer position, and may change the keymaps as described for `keymap-lookup` (see Section 23.11 [Functions for Key Lookup], page 496).

23.8 Searching the Active Keymaps

Here is a pseudo-Lisp summary of how Emacs searches the active keymaps:

```
(or (if overriding-terminal-local-map
        (find-in overriding-terminal-local-map))
    (if overriding-local-map
        (find-in overriding-local-map)
      (or (find-in (get-char-property (point) 'keymap))
          (find-in-any emulation-mode-map-alists)
          (find-in-any minor-mode-overriding-map-alist)
          (find-in-any minor-mode-map-alist)
          (if (get-char-property (point) 'local-map)
              (find-in (get-char-property (point) 'local-map))
            (find-in (current-local-map)))))
    (find-in (current-global-map)))
```

Here, *find-in* and *find-in-any* are pseudo functions that search in one keymap and in an alist of keymaps, respectively. Note that the `set-transient-map` function works by setting `overriding-terminal-local-map` (see Section 23.9 [Controlling Active Maps], page 492).

In the above pseudo-code, if a key sequence starts with a mouse event (see Section 22.7.3 [Mouse Events], page 443), that event's position is used instead of point, and the event's buffer is used instead of the current buffer. In particular, this affects how the `keymap` and `local-map` properties are looked up. If a mouse event occurs on a string embedded with a `display`, `before-string`, or `after-string` property (see Section 33.19.4 [Special Properties], page 927), and the string has a non-`nil` `keymap` or `local-map` property, that overrides the corresponding property in the underlying buffer text (i.e., the property specified by the underlying text is ignored).

When a key binding is found in one of the active keymaps, and that binding is a command, the search is over—the command is executed. However, if the binding is a symbol with a value or a string, Emacs replaces the input key sequences with the variable's value or the string, and restarts the search of the active keymaps. See Section 23.10 [Key Lookup], page 495.

The command which is finally found might also be remapped. See Section 23.14 [Remapping Commands], page 504.

23.9 Controlling the Active Keymaps

`global-map` [Variable]

This variable contains the default global keymap that maps Emacs keyboard input to commands. The global keymap is normally this keymap. The default global keymap is a full keymap that binds `self-insert-command` to all of the printing characters.

It is normal practice to change the bindings in the global keymap, but you should not assign this variable any value other than the keymap it starts out with.

`current-global-map` [Function]

This function returns the current global keymap. This is the same as the value of `global-map` unless you change one or the other. The return value is a reference, not a copy; if you use `keymap-set` or other functions on it you will alter global bindings.

```
(current-global-map)
⇒ (keymap [set-mark-command beginning-of-line ...
           delete-backward-char])
```

`current-local-map` [Function]

This function returns the current buffer's local keymap, or `nil` if it has none. In the following example, the keymap for the `*scratch*` buffer (using Lisp Interaction mode) is a sparse keymap in which the entry for `ESC`, ASCII code 27, is another sparse keymap.

```
(current-local-map)
⇒ (keymap
    (10 . eval-print-last-sexp)
    (9 . lisp-indent-line)
    (127 . backward-delete-char-untabify)
    (27 keymap
        (24 . eval-defun)
        (17 . indent-sexp)))
```

`current-local-map` returns a reference to the local keymap, not a copy of it; if you use `keymap-set` or other functions on it you will alter local bindings.

`current-minor-mode-maps` [Function]
This function returns a list of the keymaps of currently enabled minor modes.

`use-global-map` *keymap* [Function]
This function makes *keymap* the new current global keymap. It returns `nil`.

It is very unusual to change the global keymap.

`use-local-map` *keymap* [Function]
This function makes *keymap* the new local keymap of the current buffer. If *keymap* is `nil`, then the buffer has no local keymap. `use-local-map` returns `nil`. Most major mode commands use this function.

`minor-mode-map-alist` [Variable]
This variable is an alist describing keymaps that may or may not be active according to the values of certain variables. Its elements look like this:

```
(variable . keymap)
```

The keymap *keymap* is active whenever *variable* has a non-`nil` value. Typically *variable* is the variable that enables or disables a minor mode. See Section 24.3.2 [Keymaps and Minor Modes], page 547.

Note that elements of `minor-mode-map-alist` do not have the same structure as elements of `minor-mode-alist`. The map must be the CDR of the element; a list with the map as the second element will not do. The CDR can be either a keymap (a list) or a symbol whose function definition is a keymap.

When more than one minor mode keymap is active, the earlier one in `minor-mode-map-alist` takes priority. But you should design minor modes so that they don't interfere with each other. If you do this properly, the order will not matter.

See Section 24.3.2 [Keymaps and Minor Modes], page 547, for more information about minor modes. See also `minor-mode-key-binding` (see Section 23.11 [Functions for Key Lookup], page 496).

`minor-mode-overriding-map-alist` [Variable]
This variable allows major modes to override the key bindings for particular minor modes. The elements of this alist look like the elements of `minor-mode-map-alist`: `(variable . keymap)`.

If a variable appears as an element of `minor-mode-overriding-map-alist`, the map specified by that element totally replaces any map specified for the same variable in `minor-mode-map-alist`.

`minor-mode-overriding-map-alist` is automatically buffer-local in all buffers.

`overriding-local-map` [Variable]
If non-`nil`, this variable holds a keymap to use instead of the buffer's local keymap, any text property or overlay keymaps, and any minor mode keymaps. This keymap, if specified, overrides all other maps that would have been active, except for the current global map.

`overriding-terminal-local-map` [Variable]

If non-`nil`, this variable holds a keymap to use instead of `overriding-local-map`, the buffer's local keymap, text property or overlay keymaps, and all the minor mode keymaps.

This variable is always local to the current terminal and cannot be buffer-local. See Section 30.2 [Multiple Terminals], page 789. It is used to implement incremental search mode.

`overriding-local-map-menu-flag` [Variable]

If this variable is non-`nil`, the value of `overriding-local-map` or `overriding-terminal-local-map` can affect the display of the menu bar. The default value is `nil`, so those map variables have no effect on the menu bar.

Note that these two map variables do affect the execution of key sequences entered using the menu bar, even if they do not affect the menu bar display. So if a menu bar key sequence comes in, you should clear the variables before looking up and executing that key sequence. Modes that use the variables would typically do this anyway; normally they respond to events that they do not handle by "unreading" them and exiting.

`special-event-map` [Variable]

This variable holds a keymap for special events. If an event type has a binding in this keymap, then it is special, and the binding for the event is run directly by `read-event`. See Section 22.9 [Special Events], page 471.

`emulation-mode-map-alists` [Variable]

This variable holds a list of keymap alists to use for emulation modes. It is intended for modes or packages using multiple minor-mode keymaps. Each element is a keymap alist which has the same format and meaning as `minor-mode-map-alist`, or a symbol with a variable binding which is such an alist. The active keymaps in each alist are used before `minor-mode-map-alist` and `minor-mode-overriding-map-alist`.

`set-transient-map` *keymap* **&optional** *keep-pred on-exit* [Function]

This function adds *keymap* as a *transient* keymap, which takes precedence over other keymaps for one (or more) subsequent keys.

Normally, *keymap* is used just once, to look up the very next key. If the optional argument *keep-pred* is `t`, the map stays active as long as the user types keys defined in *keymap*; when the user types a key that is not in *keymap*, the transient keymap is deactivated and normal key lookup continues for that key.

The *keep-pred* argument can also be a function. In that case, the function is called with no arguments, prior to running each command, while *keymap* is active; it should return non-`nil` if *keymap* should stay active.

The optional argument *on-exit*, if non-`nil`, specifies a function that is called, with no arguments, after *keymap* is deactivated.

The optional argument *message* specifies the message to display after activating the transient map. If *message* is a string, it is the format string for the message, and any '`%k`' specifier in that string is replaced with the list of keys from the transient map.

Any other non-`nil` value of *message* stands for the default message format '`Repeat with %k`'.

If the optional argument *timeout* is non-`nil`, it should be a number that specifies how many seconds of idle time to wait before deactivating *keymap*. The value of the variable `set-transient-map-timeout`, if non-`nil`, overrides the value of this argument.

This function works by adding and removing *keymap* from the variable `overriding-terminal-local-map`, which takes precedence over all other active keymaps (see Section 23.8 [Searching Keymaps], page 491).

23.10 Key Lookup

Key lookup is the process of finding the binding of a key sequence from a given keymap. The execution or use of the binding is not part of key lookup.

Key lookup uses just the event type of each event in the key sequence; the rest of the event is ignored. In fact, a key sequence used for key lookup may designate a mouse event with just its types (a symbol) instead of the entire event (a list). See Section 22.7 [Input Events], page 441. Such a key sequence is insufficient for `command-execute` to run, but it is sufficient for looking up or rebinding a key.

When the key sequence consists of multiple events, key lookup processes the events sequentially: the binding of the first event is found, and must be a keymap; then the second event's binding is found in that keymap, and so on until all the events in the key sequence are used up. (The binding thus found for the last event may or may not be a keymap.) Thus, the process of key lookup is defined in terms of a simpler process for looking up a single event in a keymap. How that is done depends on the type of object associated with the event in that keymap.

Let's use the term *keymap entry* to describe the value found by looking up an event type in a keymap. (This doesn't include the item string and other extra elements in a keymap element for a menu item, because `keymap-lookup` and other key lookup functions don't include them in the returned value.) While any Lisp object may be stored in a keymap as a keymap entry, not all make sense for key lookup. Here is a table of the meaningful types of keymap entries:

`nil` `nil` means that the events used so far in the lookup form an undefined key. When a keymap fails to mention an event type at all, and has no default binding, that is equivalent to a binding of `nil` for that event type.

command The events used so far in the lookup form a complete key, and *command* is its binding. See Section 13.1 [What Is a Function], page 230.

array The array (either a string or a vector) is a keyboard macro. The events used so far in the lookup form a complete key, and the array is its binding. See Section 22.16 [Keyboard Macros], page 479, for more information.

keymap The events used so far in the lookup form a prefix key. The next event of the key sequence is looked up in *keymap*.

list The meaning of a list depends on what it contains:

- If the CAR of *list* is the symbol `keymap`, then the list is a keymap, and is treated as a keymap (see above).

- If the CAR of *list* is `lambda`, then the list is a lambda expression. This is presumed to be a function, and is treated as such (see above). In order to execute properly as a key binding, this function must be a command—it must have an `interactive` specification. See Section 22.2 [Defining Commands], page 425.

symbol The function definition of *symbol* is used in place of *symbol*. If that too is a symbol, then this process is repeated, any number of times. Ultimately this should lead to an object that is a keymap, a command, or a keyboard macro.

Note that keymaps and keyboard macros (strings and vectors) are not valid functions, so a symbol with a keymap, string, or vector as its function definition is invalid as a function. It is, however, valid as a key binding. If the definition is a keyboard macro, then the symbol is also valid as an argument to `command-execute` (see Section 22.3 [Interactive Call], page 433).

The symbol `undefined` is worth special mention: it means to treat the key as undefined. Strictly speaking, the key is defined, and its binding is the command `undefined`; but that command does the same thing that is done automatically for an undefined key: it rings the bell (by calling `ding`) but does not signal an error.

`undefined` is used in local keymaps to override a global key binding and make the key undefined locally. A local binding of `nil` would fail to do this because it would not override the global binding.

anything else

If any other type of object is found, the events used so far in the lookup form a complete key, and the object is its binding, but the binding is not executable as a command.

In short, a keymap entry may be a keymap, a command, a keyboard macro, a symbol that leads to one of them, or `nil`.

23.11 Functions for Key Lookup

Here are the functions and variables pertaining to key lookup.

`keymap-lookup` *keymap key* **&optional** *accept-defaults no-remap position* [Function]

This function returns the definition of *key* in *keymap*. All the other functions described in this chapter that look up keys use `keymap-lookup`. Here are examples:

```
(keymap-lookup (current-global-map) "C-x C-f")
    ⇒ find-file
(keymap-lookup (current-global-map) "C-x C-f 1 2 3 4 5")
    ⇒ 2
```

If the string or vector *key* is not a valid key sequence according to the prefix keys specified in *keymap*, it must be too long and have extra events at the end that do not fit into a single key sequence. Then the value is a number, the number of events at the front of *key* that compose a complete key.

If *accept-defaults* is non-`nil`, then `keymap-lookup` considers default bindings as well as bindings for the specific events in *key*. Otherwise, `keymap-lookup` reports only bindings for the specific sequence *key*, ignoring default bindings except when you explicitly ask about them. (To do this, supply `t` as an element of *key*; see Section 23.3 [Format of Keymaps], page 482.)

If *key* contains a meta character (not a function key), that character is implicitly replaced by a two-character sequence: the value of `meta-prefix-char`, followed by the corresponding non-meta character. Thus, the first example below is handled by conversion into the second example.

```
(keymap-lookup (current-global-map) "M-f")
    ⇒ forward-word
(keymap-lookup (current-global-map) "ESC f")
    ⇒ forward-word
```

The *keymap* argument can be `nil`, meaning to look up *key* in the current keymaps (as returned by `current-active-maps`, see Section 23.7 [Active Keymaps], page 490); or it can be a keymap or a list of keymaps, meaning to look up *key* only in the specified keymaps.

Unlike `read-key-sequence`, this function does not modify the specified events in ways that discard information (see Section 22.8.1 [Key Sequence Input], page 462). In particular, it does not convert letters to lower case and it does not change drag events to clicks.

Like the normal command loop, `keymap-lookup` will remap the command resulting from looking up *key* by looking up the command in the current keymaps. However, if the optional third argument *no-remap* is non-`nil`, `keymap-lookup` returns the command without remapping.

If the optional argument *position* is non-`nil`, it specifies a mouse position as returned by `event-start` and `event-end`, and the lookup occurs in the keymaps associated with that position, instead of in *keymap*. *position* can also be a number or a marker, in which case it is interpreted as a buffer position, and the function uses the keymap properties at that position instead of at point.

`undefined` [Command]

Used in keymaps to undefine keys. It calls `ding`, but does not cause an error.

`keymap-local-lookup` *key* **&optional** *accept-defaults* [Function]

This function returns the binding for *key* in the current local keymap, or `nil` if it is undefined there.

The argument *accept-defaults* controls checking for default bindings, as in `keymap-lookup` (above).

`keymap-global-lookup` *key* **&optional** *accept-defaults* [Function]

This function returns the binding for command *key* in the current global keymap, or `nil` if it is undefined there.

The argument *accept-defaults* controls checking for default bindings, as in `keymap-lookup` (above).

`minor-mode-key-binding` *key* **&optional** *accept-defaults* [Function]

This function returns a list of all the active minor mode bindings of *key*. More precisely, it returns an alist of pairs `(modename . binding)`, where *modename* is the variable that enables the minor mode, and *binding* is *key*'s binding in that mode. If *key* has no minor-mode bindings, the value is `nil`.

If the first binding found is not a prefix definition (a keymap or a symbol defined as a keymap), all subsequent bindings from other minor modes are omitted, since they would be completely shadowed. Similarly, the list omits non-prefix bindings that follow prefix bindings.

The argument *accept-defaults* controls checking for default bindings, as in `keymap-lookup` (above).

`meta-prefix-char` [User Option]

This variable is the meta-prefix character code. It is used for translating a meta character to a two-character sequence so it can be looked up in a keymap. For useful results, the value should be a prefix event (see Section 23.6 [Prefix Keys], page 489). The default value is 27, which is the ASCII code for `ESC`.

As long as the value of `meta-prefix-char` remains 27, key lookup translates `M-b` into `ESC b`, which is normally defined as the `backward-word` command. However, if you were to set `meta-prefix-char` to 24, the code for `C-x`, then Emacs will translate `M-b` into `C-x b`, whose standard binding is the `switch-to-buffer` command. (Don't actually do this!) Here is an illustration of what would happen:

```
meta-prefix-char                ; The default value.
     ⇒ 27
(key-binding "\M-b")
     ⇒ backward-word
?\C-x                           ; The print representation
     ⇒ 24                       ;    of a character.
(setq meta-prefix-char 24)
     ⇒ 24
(key-binding "\M-b")
     ⇒ switch-to-buffer         ; Now, typing M-b is
                                ;    like typing C-x b.

(setq meta-prefix-char 27)      ; Avoid confusion!
     ⇒ 27                       ; Restore the default value!
```

This translation of one event into two happens only for characters, not for other kinds of input events. Thus, `M-F1`, a function key, is not converted into `ESC F1`.

23.12 Changing Key Bindings

The way to rebind a key is to change its entry in a keymap. If you change a binding in the global keymap, the change is effective in all buffers (though it has no direct effect in buffers that shadow the global binding with a local one). If you change the current buffer's local map, that usually affects all buffers using the same major mode. The `keymap-global-set` and `keymap-local-set` functions are convenient interfaces for these operations (see Section 23.16 [Key Binding Commands], page 507). You can also use `keymap-set`, a more general function; then you must explicitly specify the map to change.

When choosing the key sequences for Lisp programs to rebind, please follow the Emacs conventions for use of various keys (see Section D.2 [Key Binding Conventions], page 1339).

The functions below signal an error if *keymap* is not a keymap, or if *key* is not a valid key.

key is a string representing a single key or a series of key strokes, and must satisfy `key-valid-p`. Key strokes are separated by a single space character.

Each key stroke is either a single character, or the name of an event, surrounded by angle brackets. In addition, any key stroke may be preceded by one or more modifier keys. Finally, a limited number of characters have a special shorthand syntax. Here's some example key sequences:

`f` The key `f`.

`S o m` A three key sequence of the keys `S`, `o` and `m`.

`C-c o` A two key sequence of the keys `c` with the control modifier and then the key `o`

`H-<left>` The key named `left` with the hyper modifier.

`M-RET` The `return` key with a meta modifier.

`C-M-<space>`
The `space` key with both the control and meta modifiers.

The only keys that have a special shorthand syntax are `NUL`, `RET`, `TAB`, `LFD`, `ESC`, `SPC` and `DEL`.

The modifiers have to be specified in alphabetical order: '`A-C-H-M-S-s`', which is '`Alt-Control-Hyper-Meta-Shift-super`'.

`keymap-set` *keymap key binding* [Function]
This function sets the binding for *key* in *keymap*. (If *key* is more than one event long, the change is actually made in another keymap reached from *keymap*.) The argument *binding* can be any Lisp object, but only certain types are meaningful. (For a list of meaningful types, see Section 23.10 [Key Lookup], page 495.) The value returned by `keymap-set` is *binding*.

If *key* is `<t>`, this sets the default binding in *keymap*. When an event has no binding of its own, the Emacs command loop uses the keymap's default binding, if there is one.

Every prefix of *key* must be a prefix key (i.e., bound to a keymap) or undefined; otherwise an error is signaled. If some prefix of *key* is undefined, then `keymap-set` defines it as a prefix key so that the rest of *key* can be defined as specified.

If there was previously no binding for *key* in *keymap*, the new binding is added at the beginning of *keymap*. The order of bindings in a keymap makes no difference for keyboard input, but it does matter for menu keymaps (see Section 23.18 [Menu Keymaps], page 511).

`keymap-unset` *keymap key* **&optional** *remove* [Function]
This function is the inverse of `keymap-set`, it unsets the binding for *key* in *keymap*, which is the same as setting the binding to `nil`. In order to instead remove the binding completely, specify *remove* as non-`nil`. This only makes a difference if *keymap* has a parent keymap: if you just unset a key in a child map, it will still shadow the same key in the parent keymap; using *remove* instead will allow the key in the parent keymap to be used.

Note: using `keymap-unset` with *remove* non-`nil` is intended for users to put in their init file; Emacs packages should avoid using it if possible, since they have complete control over their own keymaps anyway, and they should not be altering other packages' keymaps.

This example creates a sparse keymap and makes a number of bindings in it:

```
(setq map (make-sparse-keymap))
    ⇒ (keymap)
(keymap-set map "C-f" 'forward-char)
    ⇒ forward-char
map
    ⇒ (keymap (6 . forward-char))

;; Build sparse submap for C-x and bind f in that.
(keymap-set map "C-x f" 'forward-word)
    ⇒ forward-word
map
⇒ (keymap
    (24 keymap                ; C-x
        (102 . forward-word)) ;      f
    (6 . forward-char))       ; C-f

;; Bind C-p to the ctl-x-map.
(keymap-set map "C-p" ctl-x-map)
;; ctl-x-map
⇒ [nil ... find-file ... backward-kill-sentence]

;; Bind C-f to foo in the ctl-x-map.
(keymap-set map "C-p C-f" 'foo)
⇒ 'foo
map
⇒ (keymap     ; Note foo in ctl-x-map.
    (16 keymap [nil ... foo ... backward-kill-sentence])
    (24 keymap
        (102 . forward-word))
    (6 . forward-char))
```

Note that storing a new binding for `C-p C-f` actually works by changing an entry in `ctl-x-map`, and this has the effect of changing the bindings of both `C-p C-f` and `C-x C-f` in the default global map.

`keymap-set` is the general work horse for defining a key in a keymap. When writing modes, however, you frequently have to bind a large number of keys at once, and using `keymap-set` on them all can be tedious and error-prone. Instead you can use `define-keymap`, which creates a keymap and binds a number of keys. See Section 23.4 [Creating Keymaps], page 484, for details.

The function `substitute-key-definition` scans a keymap for keys that have a certain binding and rebinds them with a different binding. Another feature which is cleaner and can often produce the same results is to remap one command into another (see Section 23.14 [Remapping Commands], page 504).

`substitute-key-definition` *olddef newdef keymap* **&optional** *oldmap* [Function]

This function replaces *olddef* with *newdef* for any keys in *keymap* that were bound to *olddef*. In other words, *olddef* is replaced with *newdef* wherever it appears. The function returns `nil`.

For example, this redefines `C-x C-f`, if you do it in an Emacs with standard bindings:

```
(substitute-key-definition
 'find-file 'find-file-read-only (current-global-map))
```

If *oldmap* is non-`nil`, that changes the behavior of `substitute-key-definition`: the bindings in *oldmap* determine which keys to rebind. The rebindings still happen in *keymap*, not in *oldmap*. Thus, you can change one map under the control of the bindings in another. For example,

```
(substitute-key-definition
  'delete-backward-char 'my-funny-delete
  my-map global-map)
```

puts the special deletion command in `my-map` for whichever keys are globally bound to the standard deletion command.

Here is an example showing a keymap before and after substitution:

```
(setq map (list 'keymap
                (cons ?1 olddef-1)
                (cons ?2 olddef-2)
                (cons ?3 olddef-1)))
⇒ (keymap (49 . olddef-1) (50 . olddef-2) (51 . olddef-1))

(substitute-key-definition 'olddef-1 'newdef map)
⇒ nil
map
⇒ (keymap (49 . newdef) (50 . olddef-2) (51 . newdef))
```

`suppress-keymap` *keymap* **&optional** *nodigits* [Function]

This function changes the contents of the full keymap *keymap* by remapping `self-insert-command` to the command `undefined` (see Section 23.14 [Remapping Commands], page 504). This has the effect of undefining all printing characters, thus making ordinary insertion of text impossible. `suppress-keymap` returns `nil`.

If *nodigits* is `nil`, then `suppress-keymap` defines digits to run `digit-argument`, and `-` to run `negative-argument`. Otherwise it makes them undefined like the rest of the printing characters.

The `suppress-keymap` function does not make it impossible to modify a buffer, as it does not suppress commands such as `yank` and `quoted-insert`. To prevent any modification of a buffer, make it read-only (see Section 28.7 [Read Only Buffers], page 682).

Since this function modifies *keymap*, you would normally use it on a newly created keymap. Operating on an existing keymap that is used for some other purpose is likely to cause trouble; for example, suppressing `global-map` would make it impossible to use most of Emacs.

This function can be used to initialize the local keymap of a major mode for which insertion of text is not desirable. But usually such a mode should be derived from `special-mode` (see Section 24.2.5 [Basic Major Modes], page 537); then its keymap will automatically inherit from `special-mode-map`, which is already suppressed. Here is how `special-mode-map` is defined:

```
(defvar special-mode-map
  (let ((map (make-sparse-keymap)))
    (suppress-keymap map)
    (keymap-set map "q" 'quit-window)
    ...
    map))
```

23.13 Low-Level Key Binding

Historically, Emacs has supported a number of different syntaxes for defining keys. The documented way to bind a key today is to use the syntax supported by `key-valid-p`, which is what all the functions like `keymap-set` and `keymap-lookup` supports. This section documents the old-style syntax and interface functions; they should not be used in new code.

`define-key` (and other low-level functions that are used to rebind keys) understand a number of different syntaxes for the keys.

A vector containing lists of keys.
: You can use a list containing modifier names plus one base event (a character or function key name). For example, `[(control ?a) (meta b)]` is equivalent to *`C-a M-b`* and `[(hyper control left)]` is equivalent to *`C-H-left`*.

A string of characters with modifiers
: Internally, key sequences are often represented as strings using the special escape sequences for shift, control and meta modifiers (see Section 2.4.8 [String Type], page 20), but this representation can also be used by users when rebinding keys. A string like `"\M-x"` is read as containing a single *`M-x`*, `"\C-f"` is read as containing a single *`C-f`*, and `"\M-\C-x"` and `"\C-\M-x"` are both read as containing a single *`C-M-x`*.

A vector of characters and key symbols
: This is the other internal representation of key sequences. It supports a fuller range of modifiers than the string representation, and also support function keys. An example is '`[?\C-\H-x home]`', which represents the *`C-H-x`* `home` key sequence. See Section 2.4.3 [Character Type], page 11.

`define-key` *keymap key binding* **&optional** *remove* [Function]
: This function is like `keymap-set` (see Section 23.12 [Changing Key Bindings], page 498, but understands only the legacy key syntaxes.

  In addition, this function also has a *remove* argument. If it is non-`nil`, the definition will be removed. This is almost the same as setting the definition to `nil`, but makes a difference if the *keymap* has a parent, and *key* is shadowing the same binding in the parent. With *remove*, subsequent lookups will return the binding in the parent, whereas with a `nil` definition the lookups will return `nil`.

Here are other legacy key definition functions and commands, with the equivalent modern function to use instead in new code.

`global-set-key` *key binding* [Command]
: This function sets the binding of *key* in the current global map to *binding*. Use `keymap-global-set` instead.

`global-unset-key` *key* [Command]
: This function removes the binding of *key* from the current global map. Use `keymap-global-unset` instead.

`local-set-key` *key binding* [Command]
: This function sets the binding of *key* in the current local keymap to *binding*. Use `keymap-local-set` instead.

`local-unset-key` *key* [Command]
: This function removes the binding of *key* from the current local map. Use `keymap-local-unset` instead.

`substitute-key-definition` *olddef newdef keymap* **&optional** *oldmap* [Function]
: This function replaces *olddef* with *newdef* for any keys in *keymap* that were bound to *olddef*. In other words, *olddef* is replaced with *newdef* wherever it appears. The function returns `nil`. Use `keymap-substitute` instead.

`define-key-after` *map key binding* **&optional** *after* [Function]
: Define a binding in *map* for *key*, with value *binding*, just like `define-key`, but position the binding in *map* after the binding for the event *after*. The argument *key* should be of length one—a vector or string with just one element. But *after* should be a single event type—a symbol or a character, not a sequence. The new binding goes after the binding for *after*. If *after* is `t` or is omitted, then the new binding goes last, at the end of the keymap. However, new bindings are added before any inherited keymap. Use `keymap-set-after` instead of this function.

`keyboard-translate` *from to* [Function]
: This function modifies `keyboard-translate-table` to translate character code *from* into character code *to*. It creates the keyboard translate table if necessary. Use `key-translate` instead.

`key-binding` *key* **&optional** *accept-defaults no-remap position* [Function]
: This function returns the binding for *key* according to the current active keymaps. The result is `nil` if *key* is undefined in the keymaps. The argument *accept-defaults* controls checking for default bindings, as in `lookup-key` (see Section 23.11 [Functions for Key Lookup], page 496). If *no-remap* is non-`nil`, `key-binding` ignores command remappings (see Section 23.14 [Remapping Commands], page 504) and returns the binding directly specified for *key*. The optional argument *position* should be either a buffer position or an event position like the value of `event-start`; it tells the function to consult the maps determined based on that *position*.

  Emacs signals an error if *key* is not a string or a vector.

  Use `keymap-lookup` instead of this function.

`lookup-key` *keymap key* **&optional** *accept-defaults* [Function]
: This function returns the definition of *key* in *keymap*. If the string or vector *key* is not a valid key sequence according to the prefix keys specified in *keymap*, it must be too long and have extra events at the end that do not fit into a single key sequence.

Then the value is a number, the number of events at the front of *key* that compose a complete key.

If *accept-defaults* is non-`nil`, then `lookup-key` considers default bindings as well as bindings for the specific events in *key*. Otherwise, `lookup-key` reports only bindings for the specific sequence *key*, ignoring default bindings except when you explicitly ask about them.

Use `keymap-lookup` instead of this function.

`local-key-binding` *key* **&optional** *accept-defaults* [Function]

This function returns the binding for *key* in the current local keymap, or `nil` if it is undefined there.

The argument *accept-defaults* controls checking for default bindings, as in `lookup-key` (above).

`global-key-binding` *key* **&optional** *accept-defaults* [Function]

This function returns the binding for command *key* in the current global keymap, or `nil` if it is undefined there.

The argument *accept-defaults* controls checking for default bindings, as in `lookup-key` (above).

`event-convert-list` *list* [Function]

This function converts a list of modifier names and a basic event type to an event type which specifies all of them. The basic event type must be the last element of the list. For example,

```
(event-convert-list '(control ?a))
     ⇒ 1
(event-convert-list '(control meta ?a))
     ⇒ -134217727
(event-convert-list '(control super f1))
     ⇒ C-s-f1
```

23.14 Remapping Commands

A special kind of key binding can be used to *remap* one command to another, without having to refer to the key sequence(s) bound to the original command. To use this feature, make a key binding for a key sequence that starts with the dummy event `remap`, followed by the command name you want to remap; for the binding, specify the new definition (usually a command name, but possibly any other valid definition for a key binding).

For example, suppose My mode provides a special command `my-kill-line`, which should be invoked instead of `kill-line`. To establish this, its mode keymap should contain the following remapping:

```
(keymap-set my-mode-map "<remap> <kill-line>" 'my-kill-line)
```

Then, whenever `my-mode-map` is active, if the user types `C-k` (the default global key sequence for `kill-line`) Emacs will instead run `my-kill-line`.

Note that remapping only takes place through active keymaps; for example, putting a remapping in a prefix keymap like `ctl-x-map` typically has no effect, as such keymaps are

not themselves active. In addition, remapping only works through a single level; in the following example,

```
(keymap-set my-mode-map "<remap> <kill-line>" 'my-kill-line)
(keymap-set my-mode-map "<remap> <my-kill-line>" 'my-other-kill-line)
```

`kill-line` is *not* remapped to `my-other-kill-line`. Instead, if an ordinary key binding specifies `kill-line`, it is remapped to `my-kill-line`; if an ordinary binding specifies `my-kill-line`, it is remapped to `my-other-kill-line`.

To undo the remapping of a command, remap it to `nil`; e.g.,

```
(keymap-set my-mode-map "<remap> <kill-line>" nil)
```

`command-remapping` *command* **&optional** *position keymaps* [Function]
: This function returns the remapping for *command* (a symbol), given the current active keymaps. If *command* is not remapped (which is the usual situation), or not a symbol, the function returns `nil`. `position` can optionally specify a buffer position or an event position to determine the keymaps to use, as in `key-binding`.

  If the optional argument `keymaps` is non-`nil`, it specifies a list of keymaps to search in. This argument is ignored if `position` is non-`nil`.

23.15 Keymaps for Translating Sequences of Events

When the `read-key-sequence` function reads a key sequence (see Section 22.8.1 [Key Sequence Input], page 462), it uses *translation keymaps* to translate certain event sequences into others. The translation keymaps are `input-decode-map`, `local-function-key-map`, and `key-translation-map` (in order of priority).

Translation keymaps have the same structure as other keymaps, but are used differently: they specify translations to make while reading key sequences, rather than bindings for complete key sequences. As each key sequence is read, it is checked against each translation keymap. If one of the translation keymaps binds *k* to a vector *v*, then whenever *k* appears as a sub-sequence *anywhere* in a key sequence, that sub-sequence is replaced with the events in *v*.

For example, VT100 terminals send `ESC O P` when the keypad key `PF1` is pressed. On such terminals, Emacs must translate that sequence of events into a single event `pf1`. This is done by binding `ESC O P` to `[pf1]` in `input-decode-map`. Thus, when you type `C-c PF1` on the terminal, the terminal emits the character sequence `C-c ESC O P`, and `read-key-sequence` translates this back into `C-c PF1` and returns it as the vector `[?\C-c pf1]`.

Translation keymaps take effect only after Emacs has decoded the keyboard input (via the input coding system specified by `keyboard-coding-system`). See Section 34.10.8 [Terminal I/O Encoding], page 993.

`input-decode-map` [Variable]
: This variable holds a keymap that describes the character sequences sent by function keys on an ordinary character terminal.

  The value of `input-decode-map` is usually set up automatically according to the terminal's Terminfo or Termcap entry, but sometimes those need help from terminal-specific Lisp files. Emacs comes with terminal-specific files for many common terminals; their main purpose is to make entries in `input-decode-map` beyond those that

can be deduced from Termcap and Terminfo. See Section 42.1.3 [Terminal-Specific], page 1265.

`local-function-key-map` [Variable]

This variable holds a keymap similar to `input-decode-map` except that it describes key sequences which should be translated to alternative interpretations that are usually preferred. It applies after `input-decode-map` and before `key-translation-map`.

Entries in `local-function-key-map` are ignored if they conflict with bindings made in the minor mode, local, or global keymaps. I.e., the remapping only applies if the original key sequence would otherwise not have any binding.

`local-function-key-map` inherits from `function-key-map`. The latter should only be altered if you want the binding to apply in all terminals, so using the former is almost always preferred.

`key-translation-map` [Variable]

This variable is another keymap used just like `input-decode-map` to translate input events into other events. It differs from `input-decode-map` in that it goes to work after `local-function-key-map` is finished rather than before; it receives the results of translation by `local-function-key-map`.

Just like `input-decode-map`, but unlike `local-function-key-map`, this keymap is applied regardless of whether the input key-sequence has a normal binding. Note however that actual key bindings can have an effect on `key-translation-map`, even though they are overridden by it. Indeed, actual key bindings override `local-function-key-map` and thus may alter the key sequence that `key-translation-map` receives. Clearly, it is better to avoid this type of situation.

The intent of `key-translation-map` is for users to map one character set to another, including ordinary characters normally bound to `self-insert-command`.

You can use `input-decode-map`, `local-function-key-map`, and `key-translation-map` for more than simple aliases, by using a function, instead of a key sequence, as the translation of a key. Then this function is called to compute the translation of that key.

The key translation function receives one argument, which is the prompt that was specified in `read-key-sequence`—or `nil` if the key sequence is being read by the editor command loop. In most cases you can ignore the prompt value.

If the function reads input itself, it can have the effect of altering the event that follows. For example, here's how to define `C-c h` to turn the character that follows into a Hyper character:

```
(defun hyperify (prompt)
  (let ((e (read-event)))
    (vector (if (numberp e)
                (logior (ash 1 24) e)
              (if (memq 'hyper (event-modifiers e))
                  e
                (add-event-modifier "H-" e))))))

(defun add-event-modifier (string e)
  (let ((symbol (if (symbolp e) e (car e))))
    (setq symbol (intern (concat string
                                 (symbol-name symbol))))
    (if (symbolp e)
        symbol
      (cons symbol (cdr e)))))

(keymap-set local-function-key-map "C-c h" 'hyperify)
```

23.15.1 Interaction with normal keymaps

The end of a key sequence is detected when that key sequence either is bound to a command, or when Emacs determines that no additional event can lead to a sequence that is bound to a command.

This means that, while `input-decode-map` and `key-translation-map` apply regardless of whether the original key sequence would have a binding, the presence of such a binding can still prevent translation from taking place. For example, let us return to our VT100 example above and add a binding for *C-c* `ESC` to the global map; now when the user hits *C-c* `PF1` Emacs will fail to decode *C-c* `ESC` *O P* into *C-c* `PF1` because it will stop reading keys right after *C-c* `ESC`, leaving *O P* for later. This is in case the user really hit *C-c* `ESC`, in which case Emacs should not sit there waiting for the next key to decide whether the user really pressed `ESC` or `PF1`.

For that reason, it is better to avoid binding commands to key sequences where the end of the key sequence is a prefix of a key translation. The main such problematic suffixes/prefixes are `ESC`, *M-O* (which is really `ESC` *O*) and *M-[* (which is really `ESC` *[*).

23.16 Commands for Binding Keys

This section describes some convenient interactive interfaces for changing key bindings. They work by calling `keymap-set` (see Section 23.12 [Changing Key Bindings], page 498). In interactive use, these commands prompt for the argument *key* and expect the user to type a valid key sequence; they also prompt for the *binding* of the key sequence, and expect the name of a command (i.e., a symbol that satisfies `commandp`, see Section 22.3 [Interactive Call], page 433). When called from Lisp, these commands expect *key* to be a string that satisfies `key-valid-p` (see Section 23.1 [Key Sequences], page 481), and *binding* to be any Lisp object that is meaningful in a keymap (see Section 23.10 [Key Lookup], page 495).

People often use `keymap-global-set` in their init files (see Section 42.1.2 [Init File], page 1263) for simple customization. For example,

```
(keymap-global-set "C-x C-\\" 'next-line)
```

redefines `C-x C-\` to move down a line.

```
(keymap-global-set "M-<mouse-1>" 'mouse-set-point)
```

redefines the first (leftmost) mouse button, entered with the Meta key, to set point where you click.

Be careful when using non-ASCII text characters in Lisp specifications of keys to bind. If these are read as multibyte text, as they usually will be in a Lisp file (see Section 16.4 [Loading Non-ASCII], page 305), you must type the keys as multibyte too. For instance, if you use this:

```
(keymap-global-set "ö" 'my-function) ; bind o-umlaut
```

and your language environment is multibyte Latin-1, these commands actually bind the multibyte character with code 246, not the byte code 246 (`M-v`) sent by a Latin-1 terminal. In order to use this binding, you need to teach Emacs how to decode the keyboard by using an appropriate input method (see Section "Input Methods" in *The GNU Emacs Manual*).

`keymap-global-set` *key binding* [Command]

This function sets the binding of *key* in the current global map to *binding*.

```
(keymap-global-set key binding)
≡
(keymap-set (current-global-map) key binding)
```

`keymap-global-unset` *key* [Command]

This function removes the binding of *key* from the current global map.

One use of this function is in preparation for defining a longer key that uses *key* as a prefix—which would not be allowed if *key* has a non-prefix binding. For example:

```
(keymap-global-unset "C-l")
    ⇒ nil
(keymap-global-set "C-l C-l" 'redraw-display)
    ⇒ nil
```

`keymap-local-set` *key binding* [Command]

This function sets the binding of *key* in the current local keymap to *binding*.

```
(keymap-local-set key binding)
≡
(keymap-set (current-local-map) key binding)
```

`keymap-local-unset` *key* [Command]

This function removes the binding of *key* from the current local map.

23.17 Scanning Keymaps

This section describes functions used to scan all the current keymaps for the sake of printing help information. To display the bindings in a particular keymap, you can use the `describe-keymap` command (see Section "Other Help Commands" in *The GNU Emacs Manual*)

`accessible-keymaps` *keymap* **&optional** *prefix* [Function]

This function returns a list of all the keymaps that can be reached (via zero or more prefix keys) from *keymap*. The value is an association list with elements of the form `(key . map)`, where *key* is a prefix key whose definition in *keymap* is *map*.

The elements of the alist are ordered so that the *key* increases in length. The first element is always `([] . keymap)`, because the specified keymap is accessible from itself with a prefix of no events.

If *prefix* is given, it should be a prefix key sequence; then `accessible-keymaps` includes only the submaps whose prefixes start with *prefix*. These elements look just as they do in the value of `(accessible-keymaps)`; the only difference is that some elements are omitted.

In the example below, the returned alist indicates that the key `ESC`, which is displayed as '`^[`', is a prefix key whose definition is the sparse keymap `(keymap (83 . center-paragraph) (115 . foo))`.

```
(accessible-keymaps (current-local-map))
⇒(([] keymap
      (27 keymap   ; Note this keymap for ESC is repeated below.
          (83 . center-paragraph)
          (115 . center-line))
      (9 . tab-to-tab-stop))

   ("^[" keymap
    (83 . center-paragraph)
    (115 . foo)))
```

In the following example, *C-h* is a prefix key that uses a sparse keymap starting with `(keymap (118 . describe-variable)...)`. Another prefix, *C-x 4*, uses a keymap which is also the value of the variable `ctl-x-4-map`. The event `mode-line` is one of several dummy events used as prefixes for mouse actions in special parts of a window.

```
(accessible-keymaps (current-global-map))
⇒ (([] keymap [set-mark-command beginning-of-line ...
                  delete-backward-char])
    ("^H" keymap (118 . describe-variable) ...
     (8 . help-for-help))
    ("^X" keymap [x-flush-mouse-queue ...
     backward-kill-sentence])
    ("^[" keymap [mark-sexp backward-sexp ...
     backward-kill-word])
    ("^X4" keymap (15 . display-buffer) ...)
    ([mode-line] keymap
     (S-mouse-2 . mouse-split-window-horizontally) ...))
```

These are not all the keymaps you would see in actuality.

`map-keymap` *function keymap* [Function]

The function `map-keymap` calls *function* once for each binding in *keymap*. It passes two arguments, the event type and the value of the binding. If *keymap* has a parent, the parent's bindings are included as well. This works recursively: if the parent has itself a parent, then the grandparent's bindings are also included and so on.

This function is the cleanest way to examine all the bindings in a keymap.

`where-is-internal` *command* **&optional** *keymap firstonly noindirect no-remap* [Function]

This function is a subroutine used by the `where-is` command (see Section "Help" in *The GNU Emacs Manual*). It returns a list of all key sequences (of any length) that are bound to *command* in a set of keymaps.

The argument *command* can be any object; it is compared with all keymap entries using `eq`.

If *keymap* is `nil`, then the maps used are the current active keymaps, disregarding `overriding-local-map` (that is, pretending its value is `nil`). If *keymap* is a keymap, then the maps searched are *keymap* and the global keymap. If *keymap* is a list of keymaps, only those keymaps are searched.

Usually it's best to use `overriding-local-map` as the expression for *keymap*. Then `where-is-internal` searches precisely the keymaps that are active. To search only the global map, pass the value `(keymap)` (an empty keymap) as *keymap*.

If *firstonly* is `non-ascii`, then the value is a single vector representing the first key sequence found, rather than a list of all possible key sequences. If *firstonly* is `t`, then the value is the first key sequence, except that key sequences consisting entirely of ASCII characters (or meta variants of ASCII characters) are preferred to all other key sequences and that the return value can never be a menu binding.

If *noindirect* is non-`nil`, `where-is-internal` doesn't look inside menu-items to find their commands. This makes it possible to search for a menu-item itself.

The fifth argument, *no-remap*, determines how this function treats command remappings (see Section 23.14 [Remapping Commands], page 504). There are two cases of interest:

If a command *other-command* is remapped to *command*:

> If *no-remap* is `nil`, find the bindings for *other-command* and treat them as though they are also bindings for *command*. If *no-remap* is non-`nil`, include the vector `[remap` *`other-command`*`]` in the list of possible key sequences, instead of finding those bindings.

If *command* is remapped to *other-command*:

> If *no-remap* is `nil`, return the bindings for *other-command* rather than *command*. If *no-remap* is non-`nil`, return the bindings for *command*, ignoring the fact that it is remapped.

If a command maps to a key binding like `[some-event]`, and `some-event` has a symbol plist containing a non-`nil` `non-key-event` property, then that binding is ignored by `where-is-internal`.

`describe-bindings` **&optional** *prefix buffer-or-name* [Command]

This function creates a listing of all current key bindings, and displays it in a buffer named `*Help*`. The text is grouped by modes—minor modes first, then the major mode, then global bindings.

If *prefix* is non-`nil`, it should be a prefix key; then the listing includes only keys that start with *prefix*.

When several characters with consecutive ASCII codes have the same definition, they are shown together, as '*`firstchar..lastchar`*'. In this instance, you need to know the ASCII codes to understand which characters this means. For example, in the default global map, the characters '`SPC .. ~`' are described by a single line. `SPC` is ASCII 32, `~` is ASCII 126, and the characters between them include all the normal printing characters, (e.g., letters, digits, punctuation, etc.); all these characters are bound to `self-insert-command`.

If *buffer-or-name* is non-`nil`, it should be a buffer or a buffer name. Then `describe-bindings` lists that buffer's bindings, instead of the current buffer's.

23.18 Menu Keymaps

A keymap can operate as a menu as well as defining bindings for keyboard keys and mouse buttons. Menus are usually actuated with the mouse, but they can function with the keyboard also. If a menu keymap is active for the next input event, that activates the keyboard menu feature.

23.18.1 Defining Menus

A keymap acts as a menu if it has an *overall prompt string*, which is a string that appears as an element of the keymap. (See Section 23.3 [Format of Keymaps], page 482.) The string should describe the purpose of the menu's commands. Emacs displays the overall prompt string as the menu title in some cases, depending on the toolkit (if any) used for displaying menus.[1] Keyboard menus also display the overall prompt string.

The easiest way to construct a keymap with a prompt string is to specify the string as an argument when you call `make-keymap`, `make-sparse-keymap` (see Section 23.4 [Creating Keymaps], page 484), or `define-prefix-command` (see [Definition of define-prefix-command], page 490). If you do not want the keymap to operate as a menu, don't specify a prompt string for it.

`keymap-prompt` *keymap* [Function]
: This function returns the overall prompt string of *keymap*, or `nil` if it has none.

The menu's items are the bindings in the keymap. Each binding associates an event type to a definition, but the event types have no significance for the menu appearance. (Usually we use pseudo-events, symbols that the keyboard cannot generate, as the event types for menu item bindings.) The menu is generated entirely from the bindings that correspond in the keymap to these events.

The order of items in the menu is the same as the order of bindings in the keymap. Since `define-key` puts new bindings at the front, you should define the menu items starting at the bottom of the menu and moving to the top, if you care about the order. When you add an item to an existing menu, you can specify its position in the menu using `keymap-set-after` (see Section 23.18.7 [Modifying Menus], page 522).

23.18.1.1 Simple Menu Items

The simpler (and original) way to define a menu item is to bind some event type (it doesn't matter what event type) to a binding like this:

```
(item-string . real-binding)
```

The CAR, *item-string*, is the string to be displayed in the menu. It should be short—preferably one to three words. It should describe the action of the command it corresponds to. Note that not all graphical toolkits can display non-ASCII text in menus (it will work for keyboard menus and will work to a large extent with the GTK+ toolkit).

[1] It is required for menus which do not use a toolkit, e.g., on a text terminal.

You can also supply a second string, called the help string, as follows:

```
(item-string help . real-binding)
```

help specifies a help-echo string to display while the mouse is on that item in the same way as `help-echo` text properties (see [Help display], page 934).

As far as `define-key` is concerned, *item-string* and *help-string* are part of the event's binding. However, `lookup-key` returns just *real-binding*, and only *real-binding* is used for executing the key.

If *real-binding* is `nil`, then *item-string* appears in the menu but cannot be selected.

If *real-binding* is a symbol and has a non-`nil` `menu-enable` property, that property is an expression that controls whether the menu item is enabled. Every time the keymap is used to display a menu, Emacs evaluates the expression, and it enables the menu item only if the expression's value is non-`nil`. When a menu item is disabled, it is displayed in a fuzzy fashion, and cannot be selected.

The menu bar does not recalculate which items are enabled every time you look at a menu. This is because the X toolkit requires the whole tree of menus in advance. To force recalculation of the menu bar, call `force-mode-line-update` (see Section 24.4 [Mode Line Format], page 551).

23.18.1.2 Extended Menu Items

An extended-format menu item is a more flexible and also cleaner alternative to the simple format. You define an event type with a binding that's a list starting with the symbol `menu-item`. For a non-selectable string, the binding looks like this:

```
(menu-item item-name)
```

A string starting with two or more dashes specifies a separator line; see Section 23.18.1.3 [Menu Separators], page 514.

To define a real menu item which can be selected, the extended format binding looks like this:

```
(menu-item item-name real-binding
    . item-property-list)
```

Here, *item-name* is an expression which evaluates to the menu item string. Thus, the string need not be a constant.

The third element, *real-binding*, can be the command to execute (in which case you get a normal menu item). It can also be a keymap, which will result in a submenu, and *item-name* is used as the submenu name. Finally, it can be `nil`, in which case you will get a non-selectable menu item. This is mostly useful when creating separator lines and the like.

The tail of the list, *item-property-list*, has the form of a property list (see Section 5.9 [Property Lists], page 99) which contains other information.

Here is a table of the properties that are supported:

`:enable` *form*
: The result of evaluating *form* determines whether the item is enabled (non-`nil` means yes). If the item is not enabled, you can't really click on it.

`:visible` *form*
: The result of evaluating *form* determines whether the item should actually appear in the menu (non-`nil` means yes). If the item does not appear, then the menu is displayed as if this item were not defined at all.

`:help` *help*
: The value of this property, *help*, specifies a help-echo string to display while the mouse is on that item. This is displayed in the same way as `help-echo` text properties (see [Help display], page 934). Note that this must be a constant string, unlike the `help-echo` property for text and overlays.

`:button` (*type* . *selected*)
: This property provides a way to define radio buttons and toggle buttons. The CAR, *type*, says which: it should be `:toggle` or `:radio`. The CDR, *selected*, should be a form; the result of evaluating it says whether this button is currently selected.

  A *toggle* is a menu item which is labeled as either on or off according to the value of *selected*. The command itself should toggle *selected*, setting it to `t` if it is `nil`, and to `nil` if it is `t`. Here is how the menu item to toggle the `debug-on-error` flag is defined:

```
(menu-item "Debug on Error" toggle-debug-on-error
           :button (:toggle
                    . (and (boundp 'debug-on-error)
                           debug-on-error)))
```

  This works because `toggle-debug-on-error` is defined as a command which toggles the variable `debug-on-error`.

  Radio buttons are a group of menu items, in which at any time one and only one is selected. There should be a variable whose value says which one is selected at any time. The *selected* form for each radio button in the group should check whether the variable has the right value for selecting that button. Clicking on the button should set the variable so that the button you clicked on becomes selected.

`:key-sequence` *key-sequence*
: This property specifies which key sequence to display as keyboard equivalent. Before Emacs displays *key-sequence* in the menu, it verifies that *key-sequence* is really equivalent to this menu item, so it only has an effect if you specify a correct key sequence. Specifying `nil` for *key-sequence* is equivalent to the `:key-sequence` attribute being absent.

`:keys` *string*
: This property specifies that *string* is the string to display as the keyboard equivalent for this menu item. You can use the '`\\[...]`' documentation construct in *string*.

  This property can also be a function (which will be called with no arguments). This function should return a string. This function will be called every time the menu is computed, so using a function that takes a lot of time to compute is not a good idea, and it should expect to be called from any context.

`:filter` *filter-fn*
: This property provides a way to compute the menu item dynamically. The property value *filter-fn* should be a function of one argument; when it is called, its argument will be *real-binding*. The function should return the binding to use instead.

  Emacs can call this function at any time that it does redisplay or operates on menu data structures, so you should write it so it can safely be called at any time.

23.18.1.3 Menu Separators

A menu separator is a kind of menu item that doesn't display any text—instead, it divides the menu into subparts with a horizontal line. A separator looks like this in the menu keymap:

```
(menu-item separator-type)
```

where *separator-type* is a string starting with two or more dashes.

In the simplest case, *separator-type* consists of only dashes. That specifies the default kind of separator. (For compatibility, `""` and `-` also count as separators.)

Certain other values of *separator-type* specify a different style of separator. Here is a table of them:

`"--no-line"`
`"--space"`
: An extra vertical space, with no actual line.

`"--single-line"`
: A single line in the menu's foreground color.

`"--double-line"`
: A double line in the menu's foreground color.

`"--single-dashed-line"`
: A single dashed line in the menu's foreground color.

`"--double-dashed-line"`
: A double dashed line in the menu's foreground color.

`"--shadow-etched-in"`
: A single line with a 3D sunken appearance. This is the default, used separators consisting of dashes only.

`"--shadow-etched-out"`
: A single line with a 3D raised appearance.

`"--shadow-etched-in-dash"`
: A single dashed line with a 3D sunken appearance.

`"--shadow-etched-out-dash"`
: A single dashed line with a 3D raised appearance.

`"--shadow-double-etched-in"`
: Two lines with a 3D sunken appearance.

`"--shadow-double-etched-out"`
: Two lines with a 3D raised appearance.

`"--shadow-double-etched-in-dash"`
: Two dashed lines with a 3D sunken appearance.

`"--shadow-double-etched-out-dash"`
: Two dashed lines with a 3D raised appearance.

You can also give these names in another style, adding a colon after the double-dash and replacing each single dash with capitalization of the following word. Thus, `"--:singleLine"`, is equivalent to `"--single-line"`.

You can use a longer form to specify keywords such as `:enable` and `:visible` for a menu separator:

`(menu-item` *`separator-type`* `nil .` *`item-property-list`*`)`

For example:

```
(menu-item "--" nil :visible (boundp 'foo))
```

Some systems and display toolkits don't really handle all of these separator types. If you use a type that isn't supported, the menu displays a similar kind of separator that is supported.

23.18.1.4 Alias Menu Items

Sometimes it is useful to make menu items that use the same command but with different enable conditions. The best way to do this in Emacs now is with extended menu items; before that feature existed, it could be done by defining alias commands and using them in menu items. Here's an example that makes two aliases for `read-only-mode` and gives them different enable conditions:

```
(defalias 'make-read-only 'read-only-mode)
(put 'make-read-only 'menu-enable '(not buffer-read-only))
(defalias 'make-writable 'read-only-mode)
(put 'make-writable 'menu-enable 'buffer-read-only)
```

When using aliases in menus, often it is useful to display the equivalent key bindings for the real command name, not the aliases (which typically don't have any key bindings except for the menu itself). To request this, give the alias symbol a non-`nil` `menu-alias` property. Thus,

```
(put 'make-read-only 'menu-alias t)
(put 'make-writable 'menu-alias t)
```

causes menu items for `make-read-only` and `make-writable` to show the keyboard bindings for `read-only-mode`.

23.18.2 Menus and the Mouse

The usual way to make a menu keymap produce a menu is to make it the definition of a prefix key. (A Lisp program can explicitly pop up a menu and receive the user's choice—see Section 30.17 [Pop-Up Menus], page 839.)

If the prefix key ends with a mouse event, Emacs handles the menu keymap by popping up a visible menu, so that the user can select a choice with the mouse. When the user clicks

on a menu item, the event generated is whatever character or symbol has the binding that brought about that menu item. (A menu item may generate a series of events if the menu has multiple levels or comes from the menu bar.)

It's often best to use a button-down event to trigger the menu. Then the user can select a menu item by releasing the button.

If the menu keymap contains a binding to a nested keymap, the nested keymap specifies a *submenu*. There will be a menu item, labeled by the nested keymap's item string, and clicking on this item automatically pops up the specified submenu. As a special exception, if the menu keymap contains a single nested keymap and no other menu items, the menu shows the contents of the nested keymap directly, not as a submenu.

However, if Emacs is compiled without X toolkit support, or on text terminals, submenus are not supported. Each nested keymap is shown as a menu item, but clicking on it does not automatically pop up the submenu. If you wish to imitate the effect of submenus, you can do that by giving a nested keymap an item string which starts with '`@`'. This causes Emacs to display the nested keymap using a separate *menu pane*; the rest of the item string after the '`@`' is the pane label. If Emacs is compiled without X toolkit support, or if a menu is displayed on a text terminal, menu panes are not used; in that case, a '`@`' at the beginning of an item string is omitted when the menu label is displayed, and has no other effect.

23.18.3 Menus and the Keyboard

When a prefix key ending with a keyboard event (a character or function key) has a definition that is a menu keymap, the keymap operates as a keyboard menu; the user specifies the next event by choosing a menu item with the keyboard.

Emacs displays the keyboard menu with the map's overall prompt string, followed by the alternatives (the item strings of the map's bindings), in the echo area. If the bindings don't all fit at once, the user can type `SPC` to see the next line of alternatives. Successive uses of `SPC` eventually get to the end of the menu and then cycle around to the beginning. (The variable `menu-prompt-more-char` specifies which character is used for this; `SPC` is the default.)

When the user has found the desired alternative from the menu, he or she should type the corresponding character—the one whose binding is that alternative.

`menu-prompt-more-char` [Variable]

This variable specifies the character to use to ask to see the next line of a menu. Its initial value is 32, the code for `SPC`.

23.18.4 Menu Example

Here is a complete example of defining a menu keymap. It is the definition of the '`Replace`' submenu in the '`Edit`' menu in the menu bar, and it uses the extended menu item format (see Section 23.18.1.2 [Extended Menu Items], page 512). First we create the keymap, and give it a name:

```
(defvar menu-bar-replace-menu (make-sparse-keymap "Replace"))
```

Next we define the menu items:

```
(define-key menu-bar-replace-menu [tags-repl-continue]
  '(menu-item "Continue Replace" multifile-continue
              :help "Continue last tags replace operation"))
```

```
(define-key menu-bar-replace-menu [tags-repl]
  '(menu-item "Replace in tagged files" tags-query-replace
              :help "Interactively replace a regexp in all tagged files"))
(define-key menu-bar-replace-menu [separator-replace-tags]
  '(menu-item "--"))
;; ...
```

Note the symbols which the bindings are made for; these appear inside square brackets, in the key sequence being defined. In some cases, this symbol is the same as the command name; sometimes it is different. These symbols are treated as function keys, but they are not real function keys on the keyboard. They do not affect the functioning of the menu itself, but they are echoed in the echo area when the user selects from the menu, and they appear in the output of `where-is` and `apropos`.

The menu in this example is intended for use with the mouse. If a menu is intended for use with the keyboard, that is, if it is bound to a key sequence ending with a keyboard event, then the menu items should be bound to characters or real function keys, that can be typed with the keyboard.

The binding whose definition is `("--")` is a separator line. Like a real menu item, the separator has a key symbol, in this case `separator-replace-tags`. If one menu has two separators, they must have two different key symbols.

Here is how we make this menu appear as an item in the parent menu:

```
(define-key menu-bar-edit-menu [replace]
  (list 'menu-item "Replace" menu-bar-replace-menu))
```

Note that this incorporates the submenu keymap, which is the value of the variable `menu-bar-replace-menu`, rather than the symbol `menu-bar-replace-menu` itself. Using that symbol in the parent menu item would be meaningless because `menu-bar-replace-menu` is not a command.

If you wanted to attach the same replace menu to a mouse click, you can do it this way:

```
(define-key global-map [C-S-down-mouse-1]
   menu-bar-replace-menu)
```

23.18.5 The Menu Bar

Emacs usually shows a *menu bar* at the top of each frame. See Section "Menu Bars" in *The GNU Emacs Manual*. Menu bar items are subcommands of the fake function key `MENU-BAR`, as defined in the active keymaps.

To add an item to the menu bar, invent a fake function key of your own (let's call it *key*), and make a binding for the key sequence `[menu-bar key]`. Most often, the binding is a menu keymap, so that pressing a button on the menu bar item leads to another menu.

When more than one active keymap defines the same function key for the menu bar, the item appears just once. If the user clicks on that menu bar item, it brings up a single, combined menu containing all the subcommands of that item—the global subcommands, the local subcommands, and the minor mode subcommands.

The variable `overriding-local-map` is normally ignored when determining the menu bar contents. That is, the menu bar is computed from the keymaps that would be active if `overriding-local-map` were `nil`. See Section 23.7 [Active Keymaps], page 490.

Here's an example of setting up a menu bar item:

```
;; Make a menu keymap (with a prompt string)
;; and make it the menu bar item's definition.
(define-key global-map [menu-bar words]
  (cons "Words" (make-sparse-keymap "Words")))

;; Define specific subcommands in this menu.
(define-key global-map
  [menu-bar words forward]
  '("Forward word" . forward-word))
(define-key global-map
  [menu-bar words backward]
  '("Backward word" . backward-word))
```

A local keymap can cancel a menu bar item made by the global keymap by rebinding the same fake function key with `undefined` as the binding. For example, this is how Dired suppresses the '`Edit`' menu bar item:

```
(define-key dired-mode-map [menu-bar edit] 'undefined)
```

Here, `edit` is the symbol produced by a fake function key, it is used by the global map for the '`Edit`' menu bar item. The main reason to suppress a global menu bar item is to regain space for mode-specific items.

`menu-bar-final-items` [Variable]

Normally the menu bar shows global items followed by items defined by the local maps.

This variable holds a list of fake function keys for items to display at the end of the menu bar rather than in normal sequence. The default value is `(help-menu)`; thus, the '`Help`' menu item normally appears at the end of the menu bar, following local menu items.

`menu-bar-update-hook` [Variable]

This normal hook is run by redisplay to update the menu bar contents, before redisplaying the menu bar. You can use it to update menus whose contents should vary. Since this hook is run frequently, we advise you to ensure that the functions it calls do not take much time in the usual case.

Next to every menu bar item, Emacs displays a key binding that runs the same command (if such a key binding exists). This serves as a convenient hint for users who do not know the key binding. If a command has multiple bindings, Emacs normally displays the first one it finds. You can specify one particular key binding by assigning an `:advertised-binding` symbol property to the command. See Section 25.3 [Keys in Documentation], page 600.

23.18.6 Tool bars

A *tool bar* is a row of clickable icons at the top of a frame, just below the menu bar. See Section "Tool Bars" in *The GNU Emacs Manual*. Emacs normally shows a tool bar on graphical displays.

On each frame, the frame parameter `tool-bar-lines` controls how many lines' worth of height to reserve for the tool bar. A zero value suppresses the tool bar. If the value

is nonzero, and `auto-resize-tool-bars` is non-`nil`, the tool bar expands and contracts automatically as needed to hold the specified contents. If the value is `grow-only`, the tool bar expands automatically, but does not contract automatically.

The tool bar contents are controlled by a menu keymap attached to a fake function key called `TOOL-BAR` (much like the way the menu bar is controlled). So you define a tool bar item using `define-key`, like this:

```
(define-key global-map [tool-bar key] item)
```

where *key* is a fake function key to distinguish this item from other items, and *item* is a menu item key binding (see Section 23.18.1.2 [Extended Menu Items], page 512), which says how to display this item and how it behaves.

The usual menu keymap item properties, `:visible`, `:enable`, `:button`, and `:filter`, are useful in tool bar bindings and have their normal meanings. The *real-binding* in the item must be a command, not a keymap; in other words, it does not work to define a tool bar icon as a prefix key.

The `:help` property specifies a help-echo string to display while the mouse is on that item. This is displayed in the same way as `help-echo` text properties (see [Help display], page 934).

In addition, you should use the `:image` property; this is how you specify the image to display in the tool bar:

`:image` *image*

image is either a single image specification (see Section 41.17 [Images], page 1211) or a vector of four image specifications. If you use a vector of four, one of them is used, depending on circumstances:

item 0 Used when the item is enabled and selected.

item 1 Used when the item is enabled and deselected.

item 2 Used when the item is disabled and selected.

item 3 Used when the item is disabled and deselected.

The GTK+ and NS versions of Emacs ignores items 1 to 3, because disabled and/or deselected images are autocomputed from item 0.

If *image* is a single image specification, Emacs draws the tool bar button in disabled state by applying an edge-detection algorithm to the image.

The `:rtl` property specifies an alternative image to use for right-to-left languages. Only the GTK+ version of Emacs supports this at present.

Some toolkits display both an image and a text in the toolbar. If you want to force using only the image, use a `:vert-only` non-`nil` property.

Like the menu bar, the tool bar can display separators (see Section 23.18.1.3 [Menu Separators], page 514). Tool bar separators are vertical rather than horizontal, though, and only a single style is supported. They are represented in the tool bar keymap by `(menu-item "--")` entries; properties like `:visible` are not supported for tool bar separators. Separators are rendered natively in GTK+ and Nextstep tool bars; in the other cases, they are rendered using an image of a vertical line.

The default tool bar is defined so that items specific to editing do not appear for major modes whose command symbol has a `mode-class` property of `special` (see Section 24.2.1 [Major Mode Conventions], page 529). Major modes may add items to the global bar by binding `[tool-bar foo]` in their local map. It makes sense for some major modes to replace the default tool bar items completely, since not many can be accommodated conveniently, and the default bindings make this easy by using an indirection through `tool-bar-map`.

`tool-bar-map` [Variable]

By default, the global map binds `[tool-bar]` as follows:

```
(keymap-global-set "<tool-bar>"
                   `(menu-item ,(purecopy "tool bar") ignore
                               :filter tool-bar-make-keymap))
```

The function `tool-bar-make-keymap`, in turn, derives the actual tool bar map dynamically from the value of the variable `tool-bar-map`. Hence, you should normally adjust the default (global) tool bar by changing that map. Some major modes, such as Info mode, completely replace the global tool bar by making `tool-bar-map` buffer-local and setting it to a different keymap.

There are two convenience functions for defining tool bar items, as follows.

`tool-bar-add-item` *icon def key* **&rest** *props* [Function]

This function adds an item to the tool bar by modifying `tool-bar-map`. The image to use is defined by *icon*, which is the base name of an XPM, XBM or PBM image file to be located by `find-image`. Given a value '`"exit"`', say, `exit.xpm`, `exit.pbm` and `exit.xbm` would be searched for in that order on a color display. On a monochrome display, the search order is '`.pbm`', '`.xbm`' and '`.xpm`'. The binding to use is the command *def*, and *key* is the fake function key symbol in the prefix keymap. The remaining arguments *props* are additional property list elements to add to the menu item specification.

To define items in some local map, bind `tool-bar-map` with `let` around calls of this function:

```
(defvar foo-tool-bar-map
  (let ((tool-bar-map (make-sparse-keymap)))
    (tool-bar-add-item ...)
    ...
    tool-bar-map))
```

`tool-bar-add-item-from-menu` *command icon* **&optional** *map* **&rest** *props* [Function]

This function is a convenience for defining tool bar items which are consistent with existing menu bar bindings. The binding of *command* is looked up in the menu bar in *map* (default `global-map`) and modified to add an image specification for *icon*, which is found in the same way as by `tool-bar-add-item`. The resulting binding is then placed in `tool-bar-map`, so use this function only for global tool bar items.

map must contain an appropriate keymap bound to `[menu-bar]`. The remaining arguments *props* are additional property list elements to add to the menu item specification.

`tool-bar-local-item-from-menu` *command icon in-map* **&optional** *from-map* **&rest** *props* [Function]

This function is used for making non-global tool bar items. Use it like `tool-bar-add-item-from-menu` except that *in-map* specifies the local map to make the definition in. The argument *from-map* is like the *map* argument of `tool-bar-add-item-from-menu`.

`auto-resize-tool-bars` [Variable]

If this variable is non-`nil`, the tool bar automatically resizes to show all defined tool bar items—but not larger than a quarter of the frame's height.

If the value is `grow-only`, the tool bar expands automatically, but does not contract automatically. To contract the tool bar, the user has to redraw the frame by entering `C-l`.

If Emacs is built with GTK+ or Nextstep, the tool bar can only show one line, so this variable has no effect.

`auto-raise-tool-bar-buttons` [Variable]

If this variable is non-`nil`, tool bar items display in raised form when the mouse moves over them.

`tool-bar-button-margin` [Variable]

This variable specifies an extra margin to add around tool bar items. The value is an integer, a number of pixels. The default is 4.

`tool-bar-button-relief` [Variable]

This variable specifies the shadow width for tool bar items. The value is an integer, a number of pixels. The default is 1.

`tool-bar-border` [Variable]

This variable specifies the height of the border drawn below the tool bar area. An integer specifies height as a number of pixels. If the value is one of `internal-border-width` (the default) or `border-width`, the tool bar border height corresponds to the corresponding frame parameter.

You can define a special meaning for clicking on a tool bar item with the shift, control, meta, etc., modifiers. You do this by setting up additional items that relate to the original item through the fake function keys. Specifically, the additional items should use the modified versions of the same fake function key used to name the original item.

Thus, if the original item was defined this way,

```
(define-key global-map [tool-bar shell]
  '(menu-item "Shell" shell
              :image (image :type xpm :file "shell.xpm")))
```

then here is how you can define clicking on the same tool bar image with the shift modifier:

```
(define-key global-map [tool-bar S-shell] 'some-command)
```

See Section 22.7.2 [Function Keys], page 442, for more information about how to add modifiers to function keys.

If you have functions that change whether a tool bar item is enabled or not, this status is not necessarily updated visually immediately. To force recalculation of the tool bar, call `force-mode-line-update` (see Section 24.4 [Mode Line Format], page 551).

23.18.7 Modifying Menus

When you insert a new item in an existing menu, you probably want to put it in a particular place among the menu's existing items. If you use `define-key` to add the item, it normally goes at the front of the menu. To put it elsewhere in the menu, use `keymap-set-after`:

`keymap-set-after` *map key binding* **&optional** *after* [Function]

Define a binding in *map* for *key*, with value *binding*, just like `keymap-set` (see Section 23.12 [Changing Key Bindings], page 498), but position the binding in *map* after the binding for the event *after*. The argument *key* should represent a single menu item or key, and should satisfy `key-valid-p` (see Section 23.1 [Key Sequences], page 481). *after* should be a single event type—a symbol or a character, not a sequence. The new binding goes after the binding for *after*. If *after* is `t` or is omitted, then the new binding goes last, at the end of the keymap. However, new bindings are added before any inherited keymap.

Here is an example:

```
(keymap-set-after my-menu "<drink>"
  '("Drink" . drink-command) 'eat)
```

makes a binding for the fake function key `DRINK` and puts it right after the binding for `EAT`.

Here is how to insert an item called '`Work`' in the '`Signals`' menu of Shell mode, after the item `break`:

```
(keymap-set-after shell-mode-map "<menu-bar> <signals> <work>"
  '("Work" . work-command) 'break)
```

23.18.8 Easy Menu

The following macro provides a convenient way to define pop-up menus and/or menu bar menus.

`easy-menu-define` *symbol maps doc menu* [Macro]

This macro defines a pop-up menu and/or menu bar submenu, whose contents are given by *menu*.

If *symbol* is non-`nil`, it should be a symbol; then this macro defines *symbol* as a function for popping up the menu (see Section 30.17 [Pop-Up Menus], page 839), with *doc* as its documentation string. *symbol* should not be quoted.

Regardless of the value of *symbol*, if *maps* is a keymap, the menu is added to that keymap, as a top-level menu for the menu bar (see Section 23.18.5 [Menu Bar], page 517). It can also be a list of keymaps, in which case the menu is added separately to each of those keymaps.

The first element of *menu* must be a string, which serves as the menu label. It may be followed by any number of the following keyword-argument pairs:

`:filter` *function*

function must be a function which, if called with one argument—the list of the other menu items—returns the actual items to be displayed in the menu.

`:visible` *include*
: *include* is an expression; if it evaluates to `nil`, the menu is made invisible. `:included` is an alias for `:visible`.

`:active` *enable*
: *enable* is an expression; if it evaluates to `nil`, the menu is not selectable. `:enable` is an alias for `:active`.

The remaining elements in *menu* are menu items.

A menu item can be a vector of three elements, `[`*name* *callback* *enable*`]`. *name* is the menu item name (a string). *callback* is a command to run, or an expression to evaluate, when the item is chosen. *enable* is an expression; if it evaluates to `nil`, the item is disabled for selection.

Alternatively, a menu item may have the form:

```
[ name callback [ keyword arg ]... ]
```

where *name* and *callback* have the same meanings as above, and each optional *keyword* and *arg* pair should be one of the following:

`:keys` *keys*
: *keys* is a string to display as keyboard equivalent to the menu item. This is normally not needed, as keyboard equivalents are computed automatically. *keys* is expanded with `substitute-command-keys` before it is displayed (see Section 25.3 [Keys in Documentation], page 600).

`:key-sequence` *keys*
: *keys* is a hint indicating which key sequence to display as keyboard equivalent, in case the command is bound to several key sequences. It has no effect if *keys* is not bound to same command as this menu item.

`:active` *enable*
: *enable* is an expression; if it evaluates to `nil`, the item is made unselectable. `:enable` is an alias for `:active`.

`:visible` *include*
: *include* is an expression; if it evaluates to `nil`, the item is made invisible. `:included` is an alias for `:visible`.

`:label` *form*
: *form* is an expression that is evaluated to obtain a value which serves as the menu item's label (the default is *name*).

`:suffix` *form*
: *form* is an expression that is dynamically evaluated and whose value is concatenated with the menu entry's label.

`:style` *style*
: *style* is a symbol describing the type of menu item; it should be `toggle` (a checkbox), or `radio` (a radio button), or anything else (meaning an ordinary menu item).

`:selected` *selected*
: *selected* is an expression; the checkbox or radio button is selected whenever the expression's value is non-`nil`.

`:help` *help*
: *help* is a string describing the menu item.

Alternatively, a menu item can be a string. Then that string appears in the menu as unselectable text. A string consisting of dashes is displayed as a separator (see Section 23.18.1.3 [Menu Separators], page 514).

Alternatively, a menu item can be a list with the same format as *menu*. This is a submenu.

Here is an example of using `easy-menu-define` to define a menu similar to the one defined in the example in Section 23.18.5 [Menu Bar], page 517:

```
(easy-menu-define words-menu global-map
  "Menu for word navigation commands."
  '("Words"
     ["Forward word" forward-word]
     ["Backward word" backward-word]))
```

24 Major and Minor Modes

A *mode* is a set of definitions that customize Emacs behavior in useful ways. There are two varieties of modes: *minor modes*, which provide features that users can turn on and off while editing; and *major modes*, which are used for editing or interacting with a particular kind of text. Each buffer has exactly one *major mode* at a time.

This chapter describes how to write both major and minor modes, how to indicate them in the mode line, and how they run hooks supplied by the user. For related topics such as keymaps and syntax tables, see Chapter 23 [Keymaps], page 481, and Chapter 36 [Syntax Tables], page 1025.

24.1 Hooks

A *hook* is a variable where you can store a function or functions (see Section 13.1 [What Is a Function], page 230) to be called on a particular occasion by an existing program. Emacs provides hooks for the sake of customization. Most often, hooks are set up in the init file (see Section 42.1.2 [Init File], page 1263), but Lisp programs can set them also. See Appendix H [Standard Hooks], page 1403, for a list of some standard hook variables.

Most of the hooks in Emacs are *normal hooks*. These variables contain lists of functions to be called with no arguments. By convention, whenever the hook name ends in '`-hook`', that tells you it is normal. We try to make all hooks normal, as much as possible, so that you can use them in a uniform way.

Every major mode command is supposed to run a normal hook called the *mode hook* as one of the last steps of initialization. This makes it easy for a user to customize the behavior of the mode, by overriding the buffer-local variable assignments already made by the mode. Most minor mode functions also run a mode hook at the end. But hooks are used in other contexts too. For example, the hook `suspend-hook` runs just before Emacs suspends itself (see Section 42.2.2 [Suspending Emacs], page 1268).

If the hook variable's name does not end with '`-hook`', that indicates it is probably an *abnormal hook*. These differ from normal hooks in two ways: they can be called with one or more arguments, and their return values can be used in some way. The hook's documentation says how the functions are called and how their return values are used. Any functions added to an abnormal hook must follow the hook's calling convention. By convention, abnormal hook names end in '`-functions`'.

If the name of the variable ends in '`-predicate`' or '`-function`' (singular) then its value must be a function, not a list of functions. As with abnormal hooks, the expected arguments and meaning of the return value vary across such *single function hooks*. The details are explained in each variable's docstring.

Since hooks (both multi and single function) are variables, their values can be modified with `setq` or temporarily with `let`. However, it is often useful to add or remove a particular function from a hook while preserving any other functions it might have. For multi function hooks, the recommended way of doing this is with `add-hook` and `remove-hook` (see Section 24.1.2 [Setting Hooks], page 526). Most normal hook variables are initially void; `add-hook` knows how to deal with this. You can add hooks either globally or buffer-locally with `add-hook`. For hooks which hold only a single function, `add-hook` is not appropriate,

but you can use `add-function` (see Section 13.12 [Advising Functions], page 252) to combine new functions with the hook. Note that some single function hooks may be `nil` which `add-function` cannot deal with, so you must check for that before calling `add-function`.

24.1.1 Running Hooks

In this section, we document the `run-hooks` function, which is used to run a normal hook. We also document the functions for running various kinds of abnormal hooks.

`run-hooks` **&rest** *hookvars* [Function]
This function takes one or more normal hook variable names as arguments, and runs each hook in turn. Each argument should be a symbol that is a normal hook variable. These arguments are processed in the order specified.

If a hook variable has a non-`nil` value, that value should be a list of functions. `run-hooks` calls all the functions, one by one, with no arguments.

The hook variable's value can also be a single function—either a lambda expression or a symbol with a function definition—which `run-hooks` calls. But this usage is obsolete.

If the hook variable is buffer-local, the buffer-local variable will be used instead of the global variable. However, if the buffer-local variable contains the element `t`, the global hook variable will be run as well.

`run-hook-with-args` *hook* **&rest** *args* [Function]
This function runs an abnormal hook by calling all the hook functions in *hook*, passing each one the arguments *args*.

`run-hook-with-args-until-failure` *hook* **&rest** *args* [Function]
This function runs an abnormal hook by calling each hook function in turn, stopping if one of them fails by returning `nil`. Each hook function is passed the arguments *args*. If this function stops because one of the hook functions fails, it returns `nil`; otherwise it returns a non-`nil` value.

`run-hook-with-args-until-success` *hook* **&rest** *args* [Function]
This function runs an abnormal hook by calling each hook function, stopping if one of them succeeds by returning a non-`nil` value. Each hook function is passed the arguments *args*. If this function stops because one of the hook functions returns a non-`nil` value, it returns that value; otherwise it returns `nil`.

24.1.2 Setting Hooks

Here's an example that adds a function to a mode hook to turn on Auto Fill mode when in Lisp Interaction mode:

```
(add-hook 'lisp-interaction-mode-hook 'auto-fill-mode)
```

The value of a hook variable should be a list of functions. You can manipulate that list using the normal Lisp facilities, but the modular way is to use the functions `add-hook` and `remove-hook`, defined below. They take care to handle some unusual situations and avoid problems.

It works to put a `lambda`-expression function on a hook, but we recommend avoiding this because it can lead to confusion. If you add the same `lambda`-expression a second time

but write it slightly differently, you will get two equivalent but distinct functions on the hook. If you then remove one of them, the other will still be on it.

`add-hook` *hook function* **&optional** *depth local* [Function]

This function is the handy way to add function *function* to hook variable *hook*. You can use it for abnormal hooks as well as for normal hooks. *function* can be any Lisp function that can accept the proper number of arguments for *hook*. For example,

```
(add-hook 'text-mode-hook 'my-text-hook-function)
```

adds `my-text-hook-function` to the hook called `text-mode-hook`.

If *function* is already present in *hook* (comparing using `equal`), then `add-hook` does not add it a second time.

If *function* has a non-`nil` property `permanent-local-hook`, then `kill-all-local-variables` (or changing major modes) won't delete it from the hook variable's local value.

For a normal hook, hook functions should be designed so that the order in which they are executed does not matter. Any dependence on the order is asking for trouble. However, the order is predictable: normally, *function* goes at the front of the hook list, so it is executed first (barring another `add-hook` call).

In some cases, it is important to control the relative ordering of functions on the hook. The optional argument *depth* lets you indicate where the function should be inserted in the list: it should then be a number between -100 and 100 where the higher the value, the closer to the end of the list the function should go. The *depth* defaults to 0 and for backward compatibility when *depth* is a non-`nil` symbol it is interpreted as a depth of 90. Furthermore, when *depth* is strictly greater than 0 the function is added *after* rather than before functions of the same depth. One should never use a depth of 100 (or -100), because one can never be sure that no other function will ever need to come before (or after) us.

`add-hook` can handle the cases where *hook* is void or its value is a single function; it sets or changes the value to a list of functions.

If *local* is non-`nil`, that says to add *function* to the buffer-local hook list instead of to the global hook list. This makes the hook buffer-local and adds `t` to the buffer-local value. The latter acts as a flag to run the hook functions in the default value as well as in the local value.

`remove-hook` *hook function* **&optional** *local* [Function]

This function removes *function* from the hook variable *hook*. It compares *function* with elements of *hook* using `equal`, so it works for both symbols and lambda expressions.

If *local* is non-`nil`, that says to remove *function* from the buffer-local hook list instead of from the global hook list.

24.2 Major Modes

Major modes specialize Emacs for editing or interacting with particular kinds of text. Each buffer has exactly one major mode at a time. Every major mode is associated with a *major mode command*, whose name should end in '`-mode`'. This command takes care of switching

to that mode in the current buffer, by setting various buffer-local variables such as a local keymap. See Section 24.2.1 [Major Mode Conventions], page 529. Note that unlike minor modes there is no way to "turn off" a major mode, instead the buffer must be switched to a different one. However, you can temporarily *suspend* a major mode and later *restore* the suspended mode, see below.

The least specialized major mode is called *Fundamental mode*, which has no mode-specific definitions or variable settings.

`fundamental-mode` [Command]
: This is the major mode command for Fundamental mode. Unlike other mode commands, it does *not* run any mode hooks (see Section 24.2.1 [Major Mode Conventions], page 529), since you are not supposed to customize this mode.

`major-mode-suspend` [Function]
: This function works like `fundamental-mode`, in that it kills all buffer-local variables, but it also records the major mode in effect, so that it could subsequently be restored. This function and `major-mode-restore` (described next) are useful when you need to put a buffer under some specialized mode other than the one Emacs chooses for it automatically (see Section 24.2.2 [Auto Major Mode], page 532), but would also like to be able to switch back to the original mode later.

`major-mode-restore` **&optional** *avoided-modes* [Function]
: This function restores the major mode recorded by `major-mode-suspend`. If no major mode was recorded, this function calls `normal-mode` (see Section 24.2.2 [Auto Major Mode], page 532), but tries to force it not to choose any modes in *avoided-modes*, if that argument is non-`nil`.

`clean-mode` [Function]
: Changing the major mode clears out most local variables, but it doesn't remove all artifacts in the buffer (like text properties and overlays). It's rare to change a buffer from one major mode to another (except from `fundamental-mode` to everything else), so this is usually not a concern. It can sometimes be convenient (mostly when debugging a problem in a buffer) to do a "full reset" of the buffer, and that's what the `clean-mode` major mode offers. It will kill all local variables (even the permanently local ones), and also removes all overlays and text properties.

The easiest way to write a major mode is to use the macro `define-derived-mode`, which sets up the new mode as a variant of an existing major mode. See Section 24.2.4 [Derived Modes], page 535. We recommend using `define-derived-mode` even if the new mode is not an obvious derivative of another mode, as it automatically enforces many coding conventions for you. See Section 24.2.5 [Basic Major Modes], page 537, for common modes to derive from.

The standard GNU Emacs Lisp directory tree contains the code for several major modes, in files such as `text-mode.el`, `texinfo.el`, `lisp-mode.el`, and `rmail.el`. You can study these libraries to see how modes are written.

`major-mode` [User Option]

The buffer-local value of this variable holds the symbol for the current major mode. Its default value holds the default major mode for new buffers. The standard default value is `fundamental-mode`.

If the default value is `nil`, then whenever Emacs creates a new buffer via a command such as *C-x b* (`switch-to-buffer`), the new buffer is put in the major mode of the previously current buffer. As an exception, if the major mode of the previous buffer has a `mode-class` symbol property with value `special`, the new buffer is put in Fundamental mode (see Section 24.2.1 [Major Mode Conventions], page 529).

24.2.1 Major Mode Conventions

The code for every major mode should follow various coding conventions, including conventions for local keymap and syntax table initialization, function and variable names, and hooks.

If you use the `define-derived-mode` macro, it will take care of many of these conventions automatically. See Section 24.2.4 [Derived Modes], page 535. Note also that Fundamental mode is an exception to many of these conventions, because it represents the default state of Emacs.

The following list of conventions is only partial. Each major mode should aim for consistency in general with other Emacs major modes, as this makes Emacs as a whole more coherent. It is impossible to list here all the possible points where this issue might come up; if the Emacs developers point out an area where your major mode deviates from the usual conventions, please make it compatible.

- Define a major mode command whose name ends in '`-mode`'. When called with no arguments, this command should switch to the new mode in the current buffer by setting up the keymap, syntax table, and buffer-local variables in an existing buffer. It should not change the buffer's contents.
- Write a documentation string for this command that describes the special commands available in this mode. See Section 24.2.3 [Mode Help], page 535.

  The documentation string may include the special documentation substrings, '`\[`*command*`]`', '`\{`*keymap*`}`', and '`\<`*keymap*`>`', which allow the help display to adapt automatically to the user's own key bindings. See Section 25.3 [Keys in Documentation], page 600.
- The major mode command should start by calling `kill-all-local-variables`. This runs the normal hook `change-major-mode-hook`, then gets rid of the buffer-local variables of the major mode previously in effect. See Section 12.11.2 [Creating Buffer-Local], page 208.
- The major mode command should set the variable `major-mode` to the major mode command symbol. This is how `describe-mode` discovers which documentation to print.
- The major mode command should set the variable `mode-name` to the "pretty" name of the mode, usually a string (but see Section 24.4.2 [Mode Line Data], page 552, for other possible forms). The name of the mode appears in the mode line.
- Calling the major mode command twice in direct succession should not fail and should do the same thing as calling the command only once. In other words, the major mode command should be idempotent.

- Since all global names are in the same name space, all the global variables, constants, and functions that are part of the mode should have names that start with the major mode name (or with an abbreviation of it if the name is long). See Section D.1 [Coding Conventions], page 1336.
- In a major mode for editing some kind of structured text, such as a programming language, indentation of text according to structure is probably useful. So the mode should set `indent-line-function` to a suitable function, and probably customize other variables for indentation. See Section 24.7 [Auto-Indentation], page 582.
- The major mode should usually have its own keymap, which is used as the local keymap in all buffers in that mode. The major mode command should call `use-local-map` to install this local map. See Section 23.7 [Active Keymaps], page 490, for more information.

  This keymap should be stored permanently in a global variable named *`modename`*`-mode-map`. Normally the library that defines the mode sets this variable.

  See Section 12.6 [Tips for Defining], page 196, for advice about how to write the code to set up the mode's keymap variable.
- The key sequences bound in a major mode keymap should usually start with *`C-c`*, followed by a control character, a digit, or *`{`*, *`}`*, `<`, `>`, *`:`* or *`;`*. The other punctuation characters are reserved for minor modes, and ordinary letters are reserved for users.

  A major mode can also rebind the keys *`M-n`*, *`M-p`* and *`M-s`*. The bindings for *`M-n`* and *`M-p`* should normally be some kind of moving forward and backward, but this does not necessarily mean cursor motion.

  It is legitimate for a major mode to rebind a standard key sequence if it provides a command that does the same job in a way better suited to the text this mode is used for. For example, a major mode for editing a programming language might redefine *`C-M-a`* to move to the beginning of a function in a way that works better for that language. The recommended way of tailoring *`C-M-a`* to the needs of a major mode is to set `beginning-of-defun-function` (see Section 31.2.6 [List Motion], page 863) to invoke the function specific to the mode.

  It is also legitimate for a major mode to rebind a standard key sequence whose standard meaning is rarely useful in that mode. For instance, minibuffer modes rebind *`M-r`*, whose standard meaning is rarely of any use in the minibuffer. Major modes such as Dired or Rmail that do not allow self-insertion of text can reasonably redefine letters and other printing characters as special commands.
- Major modes for editing text should not define `RET` to do anything other than insert a newline. However, it is ok for specialized modes for text that users don't directly edit, such as Dired and Info modes, to redefine `RET` to do something entirely different.
- Major modes should not alter options that are primarily a matter of user preference, such as whether Auto-Fill mode is enabled. Leave this to each user to decide. However, a major mode should customize other variables so that Auto-Fill mode will work usefully *if* the user decides to use it.
- The mode may have its own syntax table or may share one with other related modes. If it has its own syntax table, it should store this in a variable named *`modename`*`-mode-syntax-table`. See Chapter 36 [Syntax Tables], page 1025.

- If the mode handles a language that has a syntax for comments, it should set the variables that define the comment syntax. See Section "Options Controlling Comments" in *The GNU Emacs Manual*.
- The mode may have its own abbrev table or may share one with other related modes. If it has its own abbrev table, it should store this in a variable named *`modename`*`-mode-abbrev-table`. If the major mode command defines any abbrevs itself, it should pass `t` for the *system-flag* argument to `define-abbrev`. See Section 38.2 [Defining Abbrevs], page 1070.
- The mode should specify how to do highlighting for Font Lock mode, by setting up a buffer-local value for the variable `font-lock-defaults` (see Section 24.6 [Font Lock Mode], page 564).
- Each face that the mode defines should, if possible, inherit from an existing Emacs face. See Section 41.12.8 [Basic Faces], page 1182, and Section 24.6.7 [Faces for Font Lock], page 574.
- Consider adding a mode-specific menu to the menu bar. This should preferably include the most important menu-specific settings and commands that will allow users discovering the main features quickly and efficiently.
- Consider adding mode-specific context menus for the mode, to be used if and when users activate the `context-menu-mode` (see Section "Menu Mouse Clicks" in *The Emacs Manual*). To this end, define a mode-specific function which builds one or more menus depending on the location of the *`mouse-3`* click in the buffer, and then add that function to the buffer-local value of `context-menu-functions`.
- The mode should specify how Imenu should find the definitions or sections of a buffer, by setting up a buffer-local value for the variable `imenu-generic-expression`, for the two variables `imenu-prev-index-position-function` and `imenu-extract-index-name-function`, or for the variable `imenu-create-index-function` (see Section 24.5 [Imenu], page 562).
- The mode can tell ElDoc mode how to retrieve different types of documentation for whatever is at point, by adding one or more buffer-local entries to the special hook `eldoc-documentation-functions`.
- The mode can specify how to complete various keywords by adding one or more buffer-local entries to the special hook `completion-at-point-functions`. See Section 21.6.8 [Completion in Buffers], page 412.
- To make a buffer-local binding for an Emacs customization variable, use `make-local-variable` in the major mode command, not `make-variable-buffer-local`. The latter function would make the variable local to every buffer in which it is subsequently set, which would affect buffers that do not use this mode. It is undesirable for a mode to have such global effects. See Section 12.11 [Buffer-Local Variables], page 207.

  With rare exceptions, the only reasonable way to use `make-variable-buffer-local` in a Lisp package is for a variable which is used only within that package. Using it on a variable used by other packages would interfere with them.
- Each major mode should have a normal *mode hook* named *`modename`*`-mode-hook`. The very last thing the major mode command should do is to call `run-mode-hooks`. This runs the normal hook `change-major-mode-after-body-hook`, the mode hook, the function `hack-local-variables` (when the buffer is visiting a file), and then

the normal hook `after-change-major-mode-hook`. See Section 24.2.6 [Mode Hooks], page 538.

- The major mode command may start by calling some other major mode command (called the *parent mode*) and then alter some of its settings. A mode that does this is called a *derived mode*. The recommended way to define one is to use the `define-derived-mode` macro, but this is not required. Such a mode should call the parent mode command inside a `delay-mode-hooks` form. (Using `define-derived-mode` does this automatically.) See Section 24.2.4 [Derived Modes], page 535, and Section 24.2.6 [Mode Hooks], page 538.
- If something special should be done if the user switches a buffer from this mode to any other major mode, this mode can set up a buffer-local value for `change-major-mode-hook` (see Section 12.11.2 [Creating Buffer-Local], page 208).
- If this mode is appropriate only for specially-prepared text produced by the mode itself (rather than by the user typing at the keyboard or by an external file), then the major mode command symbol should have a property named `mode-class` with value `special`, put on as follows:

  ```
  (put 'funny-mode 'mode-class 'special)
  ```

  This tells Emacs that new buffers created while the current buffer is in Funny mode should not be put in Funny mode, even though the default value of `major-mode` is `nil`. By default, the value of `nil` for `major-mode` means to use the current buffer's major mode when creating new buffers (see Section 24.2.2 [Auto Major Mode], page 532), but with such `special` modes, Fundamental mode is used instead. Modes such as Dired, Rmail, and Buffer List use this feature.

  The function `view-buffer` does not enable View mode in buffers whose mode-class is special, because such modes usually provide their own View-like bindings.

  The `define-derived-mode` macro automatically marks the derived mode as special if the parent mode is special. Special mode is a convenient parent for such modes to inherit from; See Section 24.2.5 [Basic Major Modes], page 537.
- If you want to make the new mode the default for files with certain recognizable names, add an element to `auto-mode-alist` to select the mode for those file names (see Section 24.2.2 [Auto Major Mode], page 532). If you define the mode command to autoload, you should add this element in the same file that calls `autoload`. If you use an autoload cookie for the mode command, you can also use an autoload cookie for the form that adds the element (see [autoload cookie], page 306). If you do not autoload the mode command, it is sufficient to add the element in the file that contains the mode definition.
- The top-level forms in the file defining the mode should be written so that they may be evaluated more than once without adverse consequences. For instance, use `defvar` or `defcustom` to set mode-related variables, so that they are not reinitialized if they already have a value (see Section 12.5 [Defining Variables], page 194).

24.2.2 How Emacs Chooses a Major Mode

When Emacs visits a file, it automatically selects a major mode for the buffer based on information in the file name or in the file itself. It also processes local variables specified in the file text.

`normal-mode` **&optional** *find-file* [Command]

This function establishes the proper major mode and buffer-local variable bindings for the current buffer. It calls `set-auto-mode` (see below). As of Emacs 26.1, it no longer runs `hack-local-variables`, this now being done in `run-mode-hooks` at the initialization of major modes (see Section 24.2.6 [Mode Hooks], page 538).

If the *find-file* argument to `normal-mode` is non-`nil`, `normal-mode` assumes that the `find-file` function is calling it. In this case, it may process local variables in the '`-*-`' line or at the end of the file. The variable `enable-local-variables` controls whether to do so. See Section "Local Variables in Files" in *The GNU Emacs Manual*, for the syntax of the local variables section of a file.

If you run `normal-mode` interactively, the argument *find-file* is normally `nil`. In this case, `normal-mode` unconditionally processes any file local variables.

The function calls `set-auto-mode` to choose and set a major mode. If this does not specify a mode, the buffer stays in the major mode determined by the default value of `major-mode` (see below).

`normal-mode` uses `condition-case` around the call to the major mode command, so errors are caught and reported as a '`File mode specification error`', followed by the original error message.

`set-auto-mode` **&optional** *keep-mode-if-same* [Function]

This function selects and sets the major mode that is appropriate for the current buffer. It bases its decision (in order of precedence) on the '`-*-`' line, on any '`mode:`' local variable near the end of a file, on the '`#!`' line (using `interpreter-mode-alist`), on the text at the beginning of the buffer (using `magic-mode-alist`), and finally on the visited file name (using `auto-mode-alist`). See Section "How Major Modes are Chosen" in *The GNU Emacs Manual*. If `enable-local-variables` is `nil`, `set-auto-mode` does not check the '`-*-`' line, or near the end of the file, for any mode tag.

There are some file types where it is not appropriate to scan the file contents for a mode specifier. For example, a tar archive may happen to contain, near the end of the file, a member file that has a local variables section specifying a mode for that particular file. This should not be applied to the containing tar file. Similarly, a tiff image file might just happen to contain a first line that seems to match the '`-*-`' pattern. For these reasons, both these file extensions are members of the list `inhibit-local-variables-regexps`. Add patterns to this list to prevent Emacs searching them for local variables of any kind (not just mode specifiers).

If *keep-mode-if-same* is non-`nil`, this function does not call the mode command if the buffer is already in the proper major mode. For instance, `set-visited-file-name` sets this to `t` to avoid killing buffer local variables that the user may have set.

`set-buffer-major-mode` *buffer* [Function]

This function sets the major mode of *buffer* to the default value of `major-mode`; if that is `nil`, it uses the current buffer's major mode (if that is suitable). As an exception, if *buffer*'s name is `*scratch*`, it sets the mode to `initial-major-mode`.

The low-level primitives for creating buffers do not use this function, but medium-level commands such as `switch-to-buffer` and `find-file-noselect` use it whenever they create buffers.

`initial-major-mode` [User Option]
The value of this variable determines the major mode of the initial `*scratch*` buffer. The value should be a symbol that is a major mode command. The default value is `lisp-interaction-mode`.

`interpreter-mode-alist` [Variable]
This variable specifies major modes to use for scripts that specify a command interpreter in a '`#!`' line. Its value is an alist with elements of the form `(`*`regexp`*` . `*`mode`*`)`; this says to use mode *mode* if the file specifies an interpreter which matches `\\`*`regexp`*`\\'`. For example, one of the default elements is `("python[0-9.]*" . python-mode)`.

`magic-mode-alist` [Variable]
This variable's value is an alist with elements of the form `(`*`regexp`*` . `*`function`*`)`, where *regexp* is a regular expression and *function* is a function or `nil`. After visiting a file, `set-auto-mode` calls *function* if the text at the beginning of the buffer matches *regexp* and *function* is non-`nil`; if *function* is `nil`, `auto-mode-alist` gets to decide the mode.

`magic-fallback-mode-alist` [Variable]
This works like `magic-mode-alist`, except that it is handled only if `auto-mode-alist` does not specify a mode for this file.

`auto-mode-alist` [Variable]
This variable contains an association list of file name patterns (regular expressions) and corresponding major mode commands. Usually, the file name patterns test for suffixes, such as '`.el`' and '`.c`', but this need not be the case. An ordinary element of the alist looks like `(`*`regexp`*` . `*`mode-function`*`)`.

For example,

```
(("\\`/tmp/fol/" . text-mode)
 ("\\.texinfo\\'" . texinfo-mode)
 ("\\.texi\\'" . texinfo-mode)
 ("\\.el\\'" . emacs-lisp-mode)
 ("\\.c\\'" . c-mode)
 ("\\.h\\'" . c-mode)
 ...)
```

When you visit a file whose expanded file name (see Section 26.9.4 [File Name Expansion], page 641), with version numbers and backup suffixes removed using `file-name-sans-versions` (see Section 26.9.1 [File Name Components], page 637), matches a *regexp*, `set-auto-mode` calls the corresponding *mode-function*. This feature enables Emacs to select the proper major mode for most files.

If an element of `auto-mode-alist` has the form `(`*`regexp function`*` t)`, then after calling *function*, Emacs searches `auto-mode-alist` again for a match against the portion of the file name that did not match before. This feature is useful for uncompression packages: an entry of the form `("\\.gz\\'" `*`function`*` t)` can uncompress the file and then put the uncompressed file in the proper mode according to the name sans '`.gz`'.

If `auto-mode-alist` has more than one element whose *regexp* matches the file name, Emacs will use the first match.

Here is an example of how to prepend several pattern pairs to `auto-mode-alist`. (You might use this sort of expression in your init file.)

```
(setq auto-mode-alist
  (append
   ;; File name (within directory) starts with a dot.
   '(("/\\.[^/]*\\'" . fundamental-mode)
     ;; File name has no dot.
     ("/[^\\./]*\\'" . fundamental-mode)
     ;; File name ends in '.C'.
     ("\\.C\\'" . c++-mode))
   auto-mode-alist))
```

24.2.3 Getting Help about a Major Mode

The `describe-mode` function provides information about major modes. It is normally bound to *C-h m*. It uses the value of the variable `major-mode` (see Section 24.2 [Major Modes], page 527), which is why every major mode command needs to set that variable.

`describe-mode` **&optional** *buffer* [Command]

This command displays the documentation of the current buffer's major mode and minor modes. It uses the `documentation` function to retrieve the documentation strings of the major and minor mode commands (see Section 25.2 [Accessing Documentation], page 598).

If called from Lisp with a non-`nil` *buffer* argument, this function displays the documentation for that buffer's major and minor modes, rather than those of the current buffer.

24.2.4 Defining Derived Modes

The recommended way to define a new major mode is to derive it from an existing one using `define-derived-mode`. If there is no closely related mode, you should inherit from either `text-mode`, `special-mode`, or `prog-mode`. See Section 24.2.5 [Basic Major Modes], page 537. If none of these are suitable, you can inherit from `fundamental-mode` (see Section 24.2 [Major Modes], page 527).

`define-derived-mode` *variant parent name docstring keyword-args. . .* *body. . .* [Macro]

This macro defines *variant* as a major mode command, using *name* as the string form of the mode name. *variant* and *parent* should be unquoted symbols.

The new command *variant* is defined to call the function *parent*, then override certain aspects of that parent mode:

- The new mode has its own sparse keymap, named *variant*`-map`. `define-derived-mode` makes the parent mode's keymap the parent of the new map, unless *variant*`-map` is already set and already has a parent.
- The new mode has its own syntax table, kept in the variable *variant*`-syntax-table`, unless you override this using the `:syntax-table` keyword (see below). `define-derived-mode` makes the parent mode's syntax-table the parent of *variant*`-syntax-table`, unless the latter is already set and already has a parent different from the standard syntax table.

- The new mode has its own abbrev table, kept in the variable `variant-abbrev-table`, unless you override this using the `:abbrev-table` keyword (see below).
- The new mode has its own mode hook, `variant-hook`. It runs this hook, after running the hooks of its ancestor modes, with `run-mode-hooks`, as the last thing it does, apart from running any `:after-hook` form it may have. See Section 24.2.6 [Mode Hooks], page 538.

In addition, you can specify how to override other aspects of *parent* with *body*. The command *variant* evaluates the forms in *body* after setting up all its usual overrides, just before running the mode hooks.

If *parent* has a non-`nil` `mode-class` symbol property, then `define-derived-mode` sets the `mode-class` property of *variant* to the same value. This ensures, for example, that if *parent* is a special mode, then *variant* is also a special mode (see Section 24.2.1 [Major Mode Conventions], page 529).

You can also specify `nil` for *parent*. This gives the new mode no parent. Then `define-derived-mode` behaves as described above, but, of course, omits all actions connected with *parent*.

The argument *docstring* specifies the documentation string for the new mode. `define-derived-mode` adds some general information about the mode's hook, followed by the mode's keymap, at the end of this documentation string. If you omit *docstring*, `define-derived-mode` generates a documentation string.

The *keyword-args* are pairs of keywords and values. The values, except for `:after-hook`'s, are evaluated. The following keywords are currently supported:

`:syntax-table`
: You can use this to explicitly specify a syntax table for the new mode. If you specify a `nil` value, the new mode uses the same syntax table as *parent*, or the standard syntax table if *parent* is `nil`. (Note that this does *not* follow the convention used for non-keyword arguments that a `nil` value is equivalent with not specifying the argument.)

`:abbrev-table`
: You can use this to explicitly specify an abbrev table for the new mode. If you specify a `nil` value, the new mode uses the same abbrev table as *parent*, or `fundamental-mode-abbrev-table` if *parent* is `nil`. (Again, a `nil` value is *not* equivalent to not specifying this keyword.)

`:interactive`
: Modes are interactive commands by default. If you specify a `nil` value, the mode defined here won't be interactive. This is useful for modes that are never meant to be activated by users manually, but are only supposed to be used in some specially-formatted buffer.

`:group`
: If this is specified, the value should be the customization group for this mode. (Not all major modes have one.) The command `customize-mode` uses this. `define-derived-mode` does *not* automatically define the specified customization group.

`:after-hook`
: This optional keyword specifies a single Lisp form to evaluate as the final act of the mode function, after the mode hooks have been run. It should not be quoted. Since the form might be evaluated after the mode function has terminated, it should not access any element of the mode function's local state. An `:after-hook` form is useful for setting up aspects of the mode which depend on the user's settings, which in turn may have been changed in a mode hook.

Here is a hypothetical example:

```
(defvar-keymap hypertext-mode-map
  "<down-mouse-3>" #'do-hyper-link)

(define-derived-mode hypertext-mode
  text-mode "Hypertext"
  "Major mode for hypertext."
  (setq-local case-fold-search nil))
```

Do not write an `interactive` spec in the definition; `define-derived-mode` does that automatically.

`derived-mode-p` **&rest** *modes* [Function]
: This function returns non-`nil` if the current major mode is derived from any of the major modes given by the symbols *modes*.

24.2.5 Basic Major Modes

Apart from Fundamental mode, there are three major modes that other major modes commonly derive from: Text mode, Prog mode, and Special mode. While Text mode is useful in its own right (e.g., for editing files ending in `.txt`), Prog mode and Special mode exist mainly to let other modes derive from them.

As far as possible, new major modes should be derived, either directly or indirectly, from one of these three modes. One reason is that this allows users to customize a single mode hook (e.g., `prog-mode-hook`) for an entire family of relevant modes (e.g., all programming language modes).

`text-mode` [Command]
: Text mode is a major mode for editing human languages. It defines the '`"`' and '`\`' characters as having punctuation syntax (see Section 36.2.1 [Syntax Class Table], page 1026), and binds `M-TAB` to `ispell-complete-word` (see Section "Spelling" in *The GNU Emacs Manual*).

  An example of a major mode derived from Text mode is HTML mode. See Section "SGML and HTML Modes" in *The GNU Emacs Manual*.

`prog-mode` [Command]
: Prog mode is a basic major mode for buffers containing programming language source code. Most of the programming language major modes built into Emacs are derived from it.

Prog mode binds `parse-sexp-ignore-comments` to `t` (see Section 36.6.1 [Motion via Parsing], page 1034) and `bidi-paragraph-direction` to `left-to-right` (see Section 41.27 [Bidirectional Display], page 1255).

`special-mode` [Command]

Special mode is a basic major mode for buffers containing text that is produced specially by Emacs, rather than directly from a file. Major modes derived from Special mode are given a `mode-class` property of `special` (see Section 24.2.1 [Major Mode Conventions], page 529).

Special mode sets the buffer to read-only. Its keymap defines several common bindings, including `q` for `quit-window` and `g` for `revert-buffer` (see Section 27.3 [Reverting], page 669).

An example of a major mode derived from Special mode is Buffer Menu mode, which is used by the `*Buffer List*` buffer. See Section "Listing Existing Buffers" in *The GNU Emacs Manual*.

In addition, modes for buffers of tabulated data can inherit from Tabulated List mode, which is in turn derived from Special mode. See Section 24.2.7 [Tabulated List Mode], page 539.

24.2.6 Mode Hooks

Every major mode command should finish by running the mode-independent normal hook `change-major-mode-after-body-hook`, its mode hook, and the normal hook `after-change-major-mode-hook`. It does this by calling `run-mode-hooks`. If the major mode is a derived mode, that is if it calls another major mode (the parent mode) in its body, it should do this inside `delay-mode-hooks` so that the parent won't run these hooks itself. Instead, the derived mode's call to `run-mode-hooks` runs the parent's mode hook too. See Section 24.2.1 [Major Mode Conventions], page 529.

Emacs versions before Emacs 22 did not have `delay-mode-hooks`. Versions before 24 did not have `change-major-mode-after-body-hook`. When user-implemented major modes do not use `run-mode-hooks` and have not been updated to use these newer features, they won't entirely follow these conventions: they may run the parent's mode hook too early, or fail to run `after-change-major-mode-hook`. If you encounter such a major mode, please correct it to follow these conventions.

When you define a major mode using `define-derived-mode`, it automatically makes sure these conventions are followed. If you define a major mode "by hand", not using `define-derived-mode`, use the following functions to handle these conventions automatically.

`run-mode-hooks` **&rest** *hookvars* [Function]

Major modes should run their mode hook using this function. It is similar to `run-hooks` (see Section 24.1 [Hooks], page 525), but it also runs `change-major-mode-after-body-hook`, `hack-local-variables` (when the buffer is visiting a file) (see Section 12.12 [File Local Variables], page 213), and `after-change-major-mode-hook`. The last thing it does is to evaluate any `:after-hook` forms declared by parent modes (see Section 24.2.4 [Derived Modes], page 535).

When this function is called during the execution of a `delay-mode-hooks` form, it does not run the hooks or `hack-local-variables` or evaluate the forms immediately. Instead, it arranges for the next call to `run-mode-hooks` to run them.

`delay-mode-hooks` *body...* [Macro]
When one major mode command calls another, it should do so inside of `delay-mode-hooks`.

This macro executes *body*, but tells all `run-mode-hooks` calls during the execution of *body* to delay running their hooks. The hooks will actually run during the next call to `run-mode-hooks` after the end of the `delay-mode-hooks` construct.

`change-major-mode-after-body-hook` [Variable]
This is a normal hook run by `run-mode-hooks`. It is run before the mode hooks.

`after-change-major-mode-hook` [Variable]
This is a normal hook run by `run-mode-hooks`. It is run at the very end of every properly-written major mode command.

24.2.7 Tabulated List mode

Tabulated List mode is a major mode for displaying tabulated data, i.e., data consisting of *entries*, each entry occupying one row of text with its contents divided into columns. Tabulated List mode provides facilities for pretty-printing rows and columns, and sorting the rows according to the values in each column. It is derived from Special mode (see Section 24.2.5 [Basic Major Modes], page 537).

Tabulated List mode is geared towards displaying text using monospaced fonts, using a single font and text size. If you want to display a table using variable pitch fonts or images, `make-vtable` can be used instead. vtable also support having more than a single table in a buffer, or having a buffer that contains both a table and additional text in it. See Section "Introduction" in `vtable`, for more information.

Tabulated List mode is intended to be used as a parent mode by a more specialized major mode. Examples include Process Menu mode (see Section 40.6 [Process Information], page 1095) and Package Menu mode (see Section "Package Menu" in *The GNU Emacs Manual*).

Such a derived mode should use `define-derived-mode` in the usual way, specifying `tabulated-list-mode` as the second argument (see Section 24.2.4 [Derived Modes], page 535). The body of the `define-derived-mode` form should specify the format of the tabulated data, by assigning values to the variables documented below; optionally, it can then call the function `tabulated-list-init-header`, which will populate a header with the names of the columns.

The derived mode should also define a *listing command*. This, not the mode command, is what the user calls (e.g., `M-x list-processes`). The listing command should create or switch to a buffer, turn on the derived mode, specify the tabulated data, and finally call `tabulated-list-print` to populate the buffer.

`tabulated-list-gui-sort-indicator-asc` [User Option]
This variable specifies the character to be used on GUI frames as an indication that the column is sorted in the ascending order.

Whenever you change the sort direction in Tabulated List buffers, this indicator toggles between ascending ("asc") and descending ("desc").

`tabulated-list-gui-sort-indicator-desc` [User Option]
Like `tabulated-list-gui-sort-indicator-asc`, but used when the column is sorted in the descending order.

`tabulated-list-tty-sort-indicator-asc` [User Option]
Like `tabulated-list-gui-sort-indicator-asc`, but used for text-mode frames.

`tabulated-list-tty-sort-indicator-desc` [User Option]
Like `tabulated-list-tty-sort-indicator-asc`, but used when the column is sorted in the descending order.

`tabulated-list-format` [Variable]
This buffer-local variable specifies the format of the Tabulated List data. Its value should be a vector. Each element of the vector represents a data column, and should be a list `(`*`name width sort . props`*`)`, where

- *name* is the column's name (a string).
- *width* is the width to reserve for the column (an integer). This is meaningless for the last column, which runs to the end of each line.
- *sort* specifies how to sort entries by the column. If `nil`, the column cannot be used for sorting. If `t`, the column is sorted by comparing string values. Otherwise, this should be a predicate function for `sort` (see Section 5.6.3 [Rearrangement], page 90), which accepts two arguments with the same form as the elements of `tabulated-list-entries` (see below).
- *props* is a plist (see Section 5.9 [Property Lists], page 99) of additional column properties. If the value of the property `:right-align` is non-`nil` then the column should be right-aligned. And the property `:pad-right` specifies the number of additional padding spaces to the right of the column (by default 1 if omitted).

`tabulated-list-entries` [Variable]
This buffer-local variable specifies the entries displayed in the Tabulated List buffer. Its value should be either a list, or a function.

If the value is a list, each list element corresponds to one entry, and should have the form `(`*`id contents`*`)`, where

- *id* is either `nil`, or a Lisp object that identifies the entry. If the latter, the cursor stays on the same entry when re-sorting entries. Comparison is done with `equal`.
- *contents* is a vector with the same number of elements as `tabulated-list-format`. Each vector element is either a string, which is inserted into the buffer as-is; an image descriptor, which is used to insert an image (see Section 41.17.2 [Image Descriptors], page 1212); or a list `(`*`label . properties`*`)`, which means to insert a text button by calling `insert-text-button` with *label* and *properties* as arguments (see Section 41.20.3 [Making Buttons], page 1238).

  There should be no newlines in any of these strings.

Otherwise, the value should be a function which returns a list of the above form when called with no arguments.

`tabulated-list-revert-hook` [Variable]
This normal hook is run prior to reverting a Tabulated List buffer. A derived mode can add a function to this hook to recompute `tabulated-list-entries`.

`tabulated-list-printer` [Variable]
The value of this variable is the function called to insert an entry at point, including its terminating newline. The function should accept two arguments, *id* and *contents*, having the same meanings as in `tabulated-list-entries`. The default value is a function which inserts an entry in a straightforward way; a mode which uses Tabulated List mode in a more complex way can specify another function.

`tabulated-list-sort-key` [Variable]
The value of this variable specifies the current sort key for the Tabulated List buffer. If it is `nil`, no sorting is done. Otherwise, it should have the form `(`*`name . flip`*`)`, where *name* is a string matching one of the column names in `tabulated-list-format`, and *flip*, if non-`nil`, means to invert the sort order.

`tabulated-list-init-header` [Function]
This function computes and sets `header-line-format` for the Tabulated List buffer (see Section 24.4.7 [Header Lines], page 560), and assigns a keymap to the header line to allow sorting entries by clicking on column headers.

Modes derived from Tabulated List mode should call this after setting the above variables (in particular, only after setting `tabulated-list-format`).

`tabulated-list-print` **&optional** *remember-pos update* [Function]
This function populates the current buffer with entries. It should be called by the listing command. It erases the buffer, sorts the entries specified by `tabulated-list-entries` according to `tabulated-list-sort-key`, then calls the function specified by `tabulated-list-printer` to insert each entry.

If the optional argument *remember-pos* is non-`nil`, this function looks for the *id* element on the current line, if any, and tries to move to that entry after all the entries are (re)inserted.

If the optional argument *update* is non-`nil`, this function will only erase or add entries that have changed since the last print. This is several times faster if most entries haven't changed since the last time this function was called. The only difference in outcome is that tags placed via `tabulated-list-put-tag` will not be removed from entries that haven't changed (normally all tags are removed).

`tabulated-list-delete-entry` [Function]
This function deletes the entry at point.

It returns a list `(`*`id cols`*`)`, where *id* is the ID of the deleted entry and *cols* is a vector of its column descriptors. It moves point to the beginning of the current line. It returns `nil` if there is no entry at point.

Note that this function only changes the buffer contents; it does not alter `tabulated-list-entries`.

`tabulated-list-get-id` **&optional** *pos* [Function]
This `defsubst` returns the ID object from `tabulated-list-entries` (if that is a list) or from the list returned by `tabulated-list-entries` (if it is a function). If omitted or `nil`, *pos* defaults to point.

`tabulated-list-get-entry` **&optional** *pos* [Function]
This `defsubst` returns the entry object from `tabulated-list-entries` (if that is a list) or from the list returned by `tabulated-list-entries` (if it is a function). This will be a vector for the ID at *pos*. If there is no entry at *pos*, then the function returns `nil`.

`tabulated-list-header-overlay-p` **&optional** *POS* [Function]
This `defsubst` returns non-`nil` if there is a fake header at *pos*. A fake header is used if `tabulated-list-use-header-line` is `nil` to put the column names at the beginning of the buffer. If omitted or `nil`, *pos* defaults to `point-min`.

`tabulated-list-put-tag` *tag* **&optional** *advance* [Function]
This function puts *tag* in the padding area of the current line. The padding area can be empty space at the beginning of the line, the width of which is governed by `tabulated-list-padding`. *tag* should be a string, with a length less than or equal to `tabulated-list-padding`. If *advance* is non-`nil`, this function advances point by one line.

`tabulated-list-clear-all-tags` [Function]
This function clears all tags from the padding area in the current buffer.

`tabulated-list-set-col` *col desc* **&optional** *change-entry-data* [Function]
This function changes the tabulated list entry at point, setting *col* to *desc*. *col* is the column number to change, or the name of the column to change. *desc* is the new column descriptor, which is inserted via `tabulated-list-print-col`.

If *change-entry-data* is non-`nil`, this function modifies the underlying data (usually the column descriptor in the list `tabulated-list-entries`) by setting the column descriptor of the vector to `desc`.

24.2.8 Generic Modes

Generic modes are simple major modes with basic support for comment syntax and Font Lock mode. To define a generic mode, use the macro `define-generic-mode`. See the file `generic-x.el` for some examples of the use of `define-generic-mode`.

`define-generic-mode` *mode comment-list keyword-list font-lock-list auto-mode-list function-list* **&optional** *docstring* [Macro]
This macro defines a generic mode command named *mode* (a symbol, not quoted). The optional argument *docstring* is the documentation for the mode command. If you do not supply it, `define-generic-mode` generates one by default.

The argument *comment-list* is a list in which each element is either a character, a string of one or two characters, or a cons cell. A character or a string is set up in the mode's syntax table as a comment starter. If the entry is a cons cell, the CAR is set up as a comment starter and the CDR as a comment ender. (Use `nil` for the latter if you

want comments to end at the end of the line.) Note that the syntax table mechanism has limitations about what comment starters and enders are actually possible. See Chapter 36 [Syntax Tables], page 1025.

The argument *keyword-list* is a list of keywords to highlight with `font-lock-keyword-face`. Each keyword should be a string. Meanwhile, *font-lock-list* is a list of additional expressions to highlight. Each element of this list should have the same form as an element of `font-lock-keywords`. See Section 24.6.2 [Search-based Fontification], page 567.

The argument *auto-mode-list* is a list of regular expressions to add to the variable `auto-mode-alist`. They are added by the execution of the `define-generic-mode` form, not by expanding the macro call.

Finally, *function-list* is a list of functions for the mode command to call for additional setup. It calls these functions just before it runs the mode hook variable *`mode`*`-hook`.

24.2.9 Major Mode Examples

Text mode is perhaps the simplest mode besides Fundamental mode. Here are excerpts from `text-mode.el` that illustrate many of the conventions listed above:

```
;; Create the syntax table for this mode.
(defvar text-mode-syntax-table
  (let ((st (make-syntax-table)))
    (modify-syntax-entry ?\" ".   " st)
    (modify-syntax-entry ?\\ ".   " st)
    ;; Add 'p' so M-c on 'hello' leads to 'Hello', not 'hello'.
    (modify-syntax-entry ?' "w p" st)
    ...
    st)
  "Syntax table used while in `text-mode'.")

;; Create the keymap for this mode.
(defvar-keymap text-mode-map
  :doc "Keymap for `text-mode'.
Many other modes, such as `mail-mode' and `outline-mode', inherit all
the commands defined in this map."
  "C-M-i" #'ispell-complete-word)
```

Here is how the actual mode command is defined now:

```
(define-derived-mode text-mode nil "Text"
  "Major mode for editing text written for humans to read.
In this mode, paragraphs are delimited only by blank or white lines.
You can thus get the full benefit of adaptive filling
 (see the variable `adaptive-fill-mode').
\\{text-mode-map}
Turning on Text mode runs the normal hook `text-mode-hook'."
  (setq-local text-mode-variant t)
  (setq-local require-final-newline mode-require-final-newline))
```

The three Lisp modes (Lisp mode, Emacs Lisp mode, and Lisp Interaction mode) have more features than Text mode and the code is correspondingly more complicated. Here are excerpts from `lisp-mode.el` that illustrate how these modes are written.

Here is how the Lisp mode syntax and abbrev tables are defined:

```
;; Create mode-specific table variables.
(define-abbrev-table 'lisp-mode-abbrev-table ()
  "Abbrev table for Lisp mode.")

(defvar lisp-mode-syntax-table
  (let ((table (make-syntax-table lisp--mode-syntax-table)))
    (modify-syntax-entry ?\[ "_   " table)
    (modify-syntax-entry ?\] "_   " table)
    (modify-syntax-entry ?# "' 14" table)
    (modify-syntax-entry ?| "\" 23bn" table)
    table)
  "Syntax table used in `lisp-mode'.")
```

The three modes for Lisp share much of their code. For instance, Lisp mode and Emacs Lisp mode inherit from Lisp Data mode and Lisp Interaction Mode inherits from Emacs Lisp mode.

Amongst other things, Lisp Data mode sets up the `comment-start` variable to handle Lisp comments:

```
(setq-local comment-start ";")
...
```

Each of the different Lisp modes has a slightly different keymap. For example, Lisp mode binds *C-c C-z* to `run-lisp`, but the other Lisp modes do not. However, all Lisp modes have some commands in common. The following code sets up the common commands:

```
(defvar-keymap lisp-mode-shared-map
  :parent prog-mode-map
  :doc "Keymap for commands shared by all sorts of Lisp modes."
  "C-M-q" #'indent-sexp
  "DEL" #'backward-delete-char-untabify)
```

And here is the code to set up the keymap for Lisp mode:

```
(defvar-keymap lisp-mode-map
  :doc "Keymap for ordinary Lisp mode.
All commands in `lisp-mode-shared-map' are inherited by this map."
  :parent lisp-mode-shared-map
  "C-M-x" #'lisp-eval-defun
  "C-c C-z" #'run-lisp)
```

Finally, here is the major mode command for Lisp mode:

```
(define-derived-mode lisp-mode lisp-data-mode "Lisp"
  "Major mode for editing Lisp code for Lisps other than GNU Emacs Lisp.
Commands:
Delete converts tabs to spaces as it moves back.
Blank lines separate paragraphs.  Semicolons start comments.

\\{lisp-mode-map}
Note that `run-lisp' may be used either to start an inferior Lisp job
or to switch back to an existing one."
  (setq-local find-tag-default-function 'lisp-find-tag-default)
  (setq-local comment-start-skip
              "\\(\\(^\\|[^\\\n]\\)\\(\\\\\\\\)*\\)\\(;+\\|#|\\) *")
  (setq imenu-case-fold-search t))
```

24.3 Minor Modes

A *minor mode* provides optional features that users may enable or disable independently of the choice of major mode. Minor modes can be enabled individually or in combination.

Most minor modes implement features that are independent of the major mode, and can thus be used with most major modes. For example, Auto Fill mode works with any major mode that permits text insertion. A few minor modes, however, are specific to a particular major mode. For example, Diff Auto Refine mode is a minor mode that is intended to be used only with Diff mode.

Ideally, a minor mode should have its desired effect regardless of the other minor modes in effect. It should be possible to activate and deactivate minor modes in any order.

`local-minor-modes` [Variable]
: This buffer-local variable lists the currently enabled minor modes in the current buffer, and is a list of symbols.

`global-minor-modes` [Variable]
: This variable lists the currently enabled global minor modes, and is a list of symbols.

`minor-mode-list` [Variable]
: The value of this variable is a list of all minor mode commands.

24.3.1 Conventions for Writing Minor Modes

There are conventions for writing minor modes just as there are for major modes (see Section 24.2 [Major Modes], page 527). These conventions are described below. The easiest way to follow them is to use the macro `define-minor-mode`. See Section 24.3.3 [Defining Minor Modes], page 547.

- Define a variable whose name ends in '`-mode`'. We call this the *mode variable*. The minor mode command should set this variable. The value will be `nil` if the mode is disabled, and non-`nil` if the mode is enabled. The variable should be buffer-local if the minor mode is buffer-local.

  This variable is used in conjunction with the `minor-mode-alist` to display the minor mode name in the mode line. It also determines whether the minor mode keymap is active, via `minor-mode-map-alist` (see Section 23.9 [Controlling Active Maps], page 492). Individual commands or hooks can also check its value.

- Define a command, called the *mode command*, whose name is the same as the mode variable. Its job is to set the value of the mode variable, plus anything else that needs to be done to actually enable or disable the mode's features.

  The mode command should accept one optional argument. If called interactively with no prefix argument, it should toggle the mode (i.e., enable if it is disabled, and disable if it is enabled). If called interactively with a prefix argument, it should enable the mode if the argument is positive and disable it otherwise.

  If the mode command is called from Lisp (i.e., non-interactively), it should enable the mode if the argument is omitted or `nil`; it should toggle the mode if the argument is the symbol `toggle`; otherwise it should treat the argument in the same way as for an interactive call with a numeric prefix argument, as described above.

  The following example shows how to implement this behavior (it is similar to the code generated by the `define-minor-mode` macro):

```
(interactive (list (or current-prefix-arg 'toggle)))
(let ((enable
```

```
      (if (eq arg 'toggle)
          (not foo-mode) ; this is the mode's mode variable
        (> (prefix-numeric-value arg) 0))))
  (if enable
      do-enable
    do-disable))
```

The reason for this somewhat complex behavior is that it lets users easily toggle the minor mode interactively, and also lets the minor mode be easily enabled in a mode hook, like this:

```
(add-hook 'text-mode-hook 'foo-mode)
```

This behaves correctly whether or not `foo-mode` was already enabled, since the `foo-mode` mode command unconditionally enables the minor mode when it is called from Lisp with no argument. Disabling a minor mode in a mode hook is a little uglier:

```
(add-hook 'text-mode-hook (lambda () (foo-mode -1)))
```

However, this is not very commonly done.

Enabling or disabling a minor mode twice in direct succession should not fail and should do the same thing as enabling or disabling it only once. In other words, the minor mode command should be idempotent.

- Add an element to `minor-mode-alist` for each minor mode (see [Definition of minor-mode-alist], page 557), if you want to indicate the minor mode in the mode line. This element should be a list of the following form:

  ```
  (mode-variable string)
  ```

  Here *mode-variable* is the variable that controls enabling of the minor mode, and *string* is a short string, starting with a space, to represent the mode in the mode line. These strings must be short so that there is room for several of them at once.

  When you add an element to `minor-mode-alist`, use `assq` to check for an existing element, to avoid duplication. For example:

  ```
  (unless (assq 'leif-mode minor-mode-alist)
    (push '(leif-mode " Leif") minor-mode-alist))
  ```

  or like this, using `add-to-list` (see Section 5.5 [List Variables], page 85):

  ```
  (add-to-list 'minor-mode-alist '(leif-mode " Leif"))
  ```

In addition, several major mode conventions (see Section 24.2.1 [Major Mode Conventions], page 529) apply to minor modes as well: those regarding the names of global symbols, the use of a hook at the end of the initialization function, and the use of keymaps and other tables.

The minor mode should, if possible, support enabling and disabling via Custom (see Chapter 15 [Customization], page 278). To do this, the mode variable should be defined with `defcustom`, usually with `:type 'boolean`. If just setting the variable is not sufficient to enable the mode, you should also specify a `:set` method which enables the mode by invoking the mode command. Note in the variable's documentation string that setting the variable other than via Custom may not take effect. Also, mark the definition with an autoload cookie (see [autoload cookie], page 306), and specify a `:require` so that customizing the variable will load the library that defines the mode. For example:

```
;;;###autoload
(defcustom msb-mode nil
  "Toggle msb-mode.
Setting this variable directly does not take effect;
use either \\[customize] or the function `msb-mode'."
  :set 'custom-set-minor-mode
  :initialize 'custom-initialize-default
  :version "20.4"
  :type    'boolean
  :group   'msb
  :require 'msb)
```

24.3.2 Keymaps and Minor Modes

Each minor mode can have its own keymap, which is active when the mode is enabled. To set up a keymap for a minor mode, add an element to the alist `minor-mode-map-alist`. See [Definition of minor-mode-map-alist], page 493.

One use of minor mode keymaps is to modify the behavior of certain self-inserting characters so that they do something else as well as self-insert. (Another way to customize `self-insert-command` is through `post-self-insert-hook`, see Section 33.5 [Commands for Insertion], page 886. Apart from this, the facilities for customizing `self-insert-command` are limited to special cases, designed for abbrevs and Auto Fill mode. Do not try substituting your own definition of `self-insert-command` for the standard one. The editor command loop handles this function specially.)

Minor modes may bind commands to key sequences consisting of `C-c` followed by a punctuation character. However, sequences consisting of `C-c` followed by one of `{}<>:;`, or a control character or digit, are reserved for major modes. Also, `C-c` *letter* is reserved for users. See Section D.2 [Key Binding Conventions], page 1339.

24.3.3 Defining Minor Modes

The macro `define-minor-mode` offers a convenient way of implementing a mode in one self-contained definition.

`define-minor-mode` *mode doc keyword-args... body...* [Macro]

This macro defines a new minor mode whose name is *mode* (a symbol). It defines a command named *mode* to toggle the minor mode, with *doc* as its documentation string.

The toggle command takes one optional (prefix) argument. If called interactively with no argument it toggles the mode on or off. A positive prefix argument enables the mode, any other prefix argument disables it. From Lisp, an argument of `toggle` toggles the mode, whereas an omitted or `nil` argument enables the mode. This makes it easy to enable the minor mode in a major mode hook, for example. If *doc* is `nil`, the macro supplies a default documentation string explaining the above.

By default, it also defines a variable named *mode*, which is set to `t` or `nil` by enabling or disabling the mode.

The *keyword-args* consist of keywords followed by corresponding values. A few keywords have special meanings:

`:global` *global*
: If non-`nil`, this specifies that the minor mode should be global rather than buffer-local. It defaults to `nil`.

  One of the effects of making a minor mode global is that the *mode* variable becomes a customization variable. Toggling it through the Customize interface turns the mode on and off, and its value can be saved for future Emacs sessions (see Section "Saving Customizations" in *The GNU Emacs Manual*. For the saved variable to work, you should ensure that the minor mode function is available each time Emacs starts; usually this is done by marking the `define-minor-mode` form as autoloaded.

`:init-value` *init-value*
: This is the value to which the *mode* variable is initialized. Except in unusual circumstances (see below), this value must be `nil`.

`:lighter` *lighter*
: The string *lighter* says what to display in the mode line when the mode is enabled; if it is `nil`, the mode is not displayed in the mode line.

`:keymap` *keymap*
: The optional argument *keymap* specifies the keymap for the minor mode. If non-`nil`, it should be a variable name (whose value is a keymap), a keymap, or an alist of the form

  ```
  (key-sequence . definition)
  ```

  where each *key-sequence* and *definition* are arguments suitable for passing to `define-key` (see Section 23.12 [Changing Key Bindings], page 498). If *keymap* is a keymap or an alist, this also defines the variable *mode*`-map`.

`:variable` *place*
: This replaces the default variable *mode*, used to store the state of the mode. If you specify this, the *mode* variable is not defined, and any *init-value* argument is unused. *place* can be a different named variable (which you must define yourself), or anything that can be used with the `setf` function (see Section 12.17 [Generalized Variables], page 224). *place* can also be a cons `(`*get* `.` *set*`)`, where *get* is an expression that returns the current state, and *set* is a function of one argument (a state) which should be assigned to *place*.

`:after-hook` *after-hook*
: This defines a single Lisp form which is evaluated after the mode hooks have run. It should not be quoted.

`:interactive` *value*
: Minor modes are interactive commands by default. If *value* is `nil`, this is inhibited. If *value* is a list of symbols, it's used to say which major modes this minor mode is useful in.

Any other keyword arguments are passed directly to the `defcustom` generated for the variable *mode*. See Section 15.3 [Variable Definitions], page 281, for the description of those keywords and their values.

> The command named *mode* first performs the standard actions such as setting the variable named *mode* and then executes the *body* forms, if any. It then runs the mode hook variable *mode*`-hook` and finishes by evaluating any form in `:after-hook`. (Note that all of this, including running the hook, is done both when the mode is enabled and disabled.)

The initial value must be `nil` except in cases where (1) the mode is preloaded in Emacs, or (2) it is painless for loading to enable the mode even though the user did not request it. For instance, if the mode has no effect unless something else is enabled, and will always be loaded by that time, enabling it by default is harmless. But these are unusual circumstances. Normally, the initial value must be `nil`.

The name `easy-mmode-define-minor-mode` is an alias for this macro.

Here is an example of using `define-minor-mode`:

```
(define-minor-mode hungry-mode
  "Toggle Hungry mode.
Interactively with no argument, this command toggles the mode.
A positive prefix argument enables the mode, any other prefix
argument disables it.  From Lisp, argument omitted or nil enables
the mode, `toggle' toggles the state.

When Hungry mode is enabled, the control delete key
gobbles all preceding whitespace except the last.
See the command \\[hungry-electric-delete]."
 ;; The initial value.
 nil
 ;; The indicator for the mode line.
 " Hungry"
 ;; The minor mode bindings.
 '(([C-backspace] . hungry-electric-delete)))
```

This defines a minor mode named “Hungry mode”, a command named `hungry-mode` to toggle it, a variable named `hungry-mode` which indicates whether the mode is enabled, and a variable named `hungry-mode-map` which holds the keymap that is active when the mode is enabled. It initializes the keymap with a key binding for *C*`-DEL`. There are no *body* forms—many minor modes don’t need any.

Here’s an equivalent way to write it:

```
(define-minor-mode hungry-mode
  "Toggle Hungry mode.
...rest of documentation as before..."
 ;; The initial value.
 :init-value nil
 ;; The indicator for the mode line.
 :lighter " Hungry"
 ;; The minor mode bindings.
 :keymap
 '(([C-backspace] . hungry-electric-delete)
   ([C-M-backspace]
    . (lambda ()
        (interactive)
        (hungry-electric-delete t)))))
```

`define-globalized-minor-mode` *global-mode mode turn-on keyword-args... body...* [Macro]

This defines a global toggle named *global-mode* whose meaning is to enable or disable the buffer-local minor mode *mode* in all (or some; see below) buffers. It also executes the *body* forms. To turn on the minor mode in a buffer, it uses the function *turn-on*; to turn off the minor mode, it calls *mode* with -1 as argument. (The function *turn-on* is a separate function so it could determine whether to enable the minor mode or not when it is not a priori clear that it should always be enabled.)

Globally enabling the mode also affects buffers subsequently created by visiting files, and buffers that use a major mode other than Fundamental mode; but it does not detect the creation of a new buffer in Fundamental mode.

This macro defines the customization option *global-mode* (see Chapter 15 [Customization], page 278), which can be toggled via the Customize interface to turn the minor mode on and off. As with `define-minor-mode`, you should ensure that the `define-globalized-minor-mode` form is evaluated each time Emacs starts, for example by providing a `:require` keyword.

Use `:group` *group* in *keyword-args* to specify the custom group for the mode variable of the global minor mode.

By default, the buffer-local minor mode variable that says whether the mode is switched on or off is the same as the name of the mode itself. Use `:variable` *variable* if that's not the case–some minor modes use a different variable to store this state information.

Generally speaking, when you define a globalized minor mode, you should also define a non-globalized version, so that people could use it (or disable it) in individual buffers. This also allows them to disable a globally enabled minor mode in a specific major mode, by using that mode's hook.

If the macro is given a `:predicate` keyword, it will create a user option called the same as the global mode variable, but with `-modes` instead of `-mode` at the end, i.e. *global-modes*. This variable will be used in a predicate function that determines whether the minor mode should be activated in a particular major mode, and users can customize the value of the variable to control the modes in which the minor mode will be switched on. Valid values of `:predicate` (and thus valid values of the user option it creates) include `t` (use in all major modes), `nil` (don't use in any major modes), or a list of mode names, optionally preceded with `not` (as in `(not` *mode-name* `...)`). These elements can be mixed, as shown in the following examples.

```
(c-mode (not mail-mode message-mode) text-mode)
```

This means "use in modes derived from `c-mode`, and not in modes derived from `message-mode` or `mail-mode`, but do use in modes derived from `text-mode`, and otherwise no other modes".

```
((not c-mode) t)
```

This means "don't use in modes derived from `c-mode`, but do use everywhere else".

```
(text-mode)
```

This means "use in modes derived from `text-mode`, but nowhere else". (There's an implicit `nil` element at the end.)

`buffer-local-set-state` *variable value...* [Macro]

Minor modes often set buffer-local variables that affect some features in Emacs. When a minor mode is switched off, the mode is expected to restore the previous state of these variables. This convenience macro helps with doing that: It works much like `setq-local`, but returns an object that can be used to restore these values back to their previous values/states (using the companion function `buffer-local-restore-state`).

24.4 Mode Line Format

Each Emacs window (aside from minibuffer windows) typically has a mode line at the bottom, which displays status information about the buffer displayed in the window. The mode line contains information about the buffer, such as its name, associated file, depth of recursive editing, and major and minor modes. A window can also have a *header line*, which is much like the mode line but appears at the top of the window.

This section describes how to control the contents of the mode line and header line. We include it in this chapter because much of the information displayed in the mode line relates to the enabled major and minor modes.

24.4.1 Mode Line Basics

The contents of each mode line are specified by the buffer-local variable `mode-line-format` (see Section 24.4.3 [Mode Line Top], page 553). This variable holds a *mode line construct*: a template that controls what is displayed on the buffer's mode line. The value of `header-line-format` specifies the buffer's header line in the same way. All windows for the same buffer use the same `mode-line-format` and `header-line-format` unless a `mode-line-format` or `header-line-format` parameter has been specified for that window (see Section 29.27 [Window Parameters], page 779).

For efficiency, Emacs does not continuously recompute each window's mode line and header line. It does so when circumstances appear to call for it—for instance, if you change the window configuration, switch buffers, narrow or widen the buffer, scroll, or modify the buffer. If you alter any of the variables referenced by `mode-line-format` or `header-line-format` (see Section 24.4.4 [Mode Line Variables], page 555), or any other data structures that affect how text is displayed (see Chapter 41 [Display], page 1133), you should use the function `force-mode-line-update` to update the display.

`force-mode-line-update` **&optional** *all* [Function]

This function forces Emacs to update the current buffer's mode line and header line, based on the latest values of all relevant variables, during its next redisplay cycle. If the optional argument *all* is non-`nil`, it forces an update for all mode lines and header lines.

This function also forces an update of the menu bar and frame title.

The selected window's mode line is usually displayed in a different color using the face `mode-line-active`. Other windows' mode lines appear in the face `mode-line-inactive` instead. See Section 41.12 [Faces], page 1168.

`mode-line-window-selected-p` [Function]

If you want to have more extensive differences between the mode lines in selected and non-selected windows, you can use this predicate in an `:eval` construct. For instance, if you want to display the buffer name in bold in selected windows, but in italics in the other windows, you can say something like:

```
(setq-default
 mode-line-buffer-identification
 '(:eval (propertize "%12b"
     'face (if (mode-line-window-selected-p)
       'bold
     'italic))))
```

Some modes put a lot of data in the mode line, pushing elements at the end of the mode line off to the right. Emacs can "compress" the mode line if the `mode-line-compact` variable is non-`nil` by turning stretches of spaces into a single space. If this variable is `long`, this is only done when the mode line is wider than the currently selected window. (This computation is approximate, based on the number of characters, and not their displayed width.) This variable can be buffer-local to only compress mode-lines in certain buffers.

24.4.2 The Data Structure of the Mode Line

The mode line contents are controlled by a data structure called a *mode line construct*, made up of lists, strings, symbols, and numbers kept in buffer-local variables. Each data type has a specific meaning for the mode line appearance, as described below. The same data structure is used for constructing frame titles (see Section 30.6 [Frame Titles], page 823) and header lines (see Section 24.4.7 [Header Lines], page 560).

A mode line construct may be as simple as a fixed string of text, but it usually specifies how to combine fixed strings with variables' values to construct the text. Many of these variables are themselves defined to have mode line constructs as their values.

Here are the meanings of various data types as mode line constructs:

`string` A string as a mode line construct appears verbatim except for `%`-*constructs* in it. These stand for substitution of other data; see Section 24.4.5 [%-Constructs], page 557.

If parts of the string have `face` properties, they control display of the text just as they would text in the buffer. Any characters which have no `face` properties are displayed, by default, in the face `mode-line` or `mode-line-inactive` (see Section "Standard Faces" in *The GNU Emacs Manual*). The `help-echo` and `keymap` properties in *string* have special meanings. See Section 24.4.6 [Properties in Mode], page 559.

`symbol` A symbol as a mode line construct stands for its value. The value of *symbol* is used as a mode line construct, in place of *symbol*. However, the symbols `t` and `nil` are ignored, as is any symbol whose value is void.

There is one exception: if the value of *symbol* is a string, it is displayed verbatim: the `%`-constructs are not recognized.

Unless *symbol* is marked as risky (i.e., it has a non-`nil` `risky-local-variable` property), all text properties specified in *symbol*'s value are ignored. This

includes the text properties of strings in *symbol*'s value, as well as all `:eval` and `:propertize` forms in it. (The reason for this is security: non-risky variables could be set automatically from file variables without prompting the user.)

`(`*`string rest...`*`)`
`(`*`list rest...`*`)`
A list whose first element is a string or list means to process all the elements recursively and concatenate the results. This is the most common form of mode line construct. (Note that text properties are handled specially (for reasons of efficiency) when displaying strings in the mode line: Only the text property on the first character of the string are considered, and they are then used over the entire string. If you need a string with different text properties, you have to use the special `:propertize` mode line construct.)

`(:eval` *`form`*`)`
A list whose first element is the symbol `:eval` says to evaluate *form*, and use the result as a string to display. Make sure this evaluation cannot load any files, as doing so could cause infinite recursion.

`(:propertize` *`elt props...`*`)`
A list whose first element is the symbol `:propertize` says to process the mode line construct *elt* recursively, then add the text properties specified by *props* to the result. The argument *props* should consist of zero or more pairs *text-property value*. If *elt* is or produces a string with text properties, all the characters of that string should have the same properties, or else some of them might be removed by `:propertize`.

`(`*`symbol then else`*`)`
A list whose first element is a symbol that is not a keyword specifies a conditional. Its meaning depends on the value of *symbol*. If *symbol* has a non-`nil` value, the second element, *then*, is processed recursively as a mode line construct. Otherwise, the third element, *else*, is processed recursively. You may omit *else*; then the mode line construct displays nothing if the value of *symbol* is `nil` or void.

`(`*`width rest...`*`)`
A list whose first element is an integer specifies truncation or padding of the results of *rest*. The remaining elements *rest* are processed recursively as mode line constructs and concatenated together. When *width* is positive, the result is space filled on the right if its width is less than *width*. When *width* is negative, the result is truncated on the right to $-$*width* columns if its width exceeds $-$*width*.

For example, the usual way to show what percentage of a buffer is above the top of the window is to use a list like this: `(-3 "%p")`.

24.4.3 The Top Level of Mode Line Control

The variable in overall control of the mode line is `mode-line-format`.

`mode-line-format` [User Option]

The value of this variable is a mode line construct that controls the contents of the mode-line. It is always buffer-local in all buffers.

If you set this variable to `nil` in a buffer, that buffer does not have a mode line. (A window that is just one line tall also does not display a mode line.)

The default value of `mode-line-format` is designed to use the values of other variables such as `mode-line-position` and `mode-line-modes` (which in turn incorporates the values of the variables `mode-name` and `minor-mode-alist`). Very few modes need to alter `mode-line-format` itself. For most purposes, it is sufficient to alter some of the variables that `mode-line-format` either directly or indirectly refers to.

If you do alter `mode-line-format` itself, the new value should use the same variables that appear in the default value (see Section 24.4.4 [Mode Line Variables], page 555), rather than duplicating their contents or displaying the information in another fashion. This way, customizations made by the user or by Lisp programs (such as `display-time` and major modes) via changes to those variables remain effective.

Here is a hypothetical example of a `mode-line-format` that might be useful for Shell mode (in reality, Shell mode does not set `mode-line-format`):

```
(setq mode-line-format
  (list "-"
   'mode-line-mule-info
   'mode-line-modified
   'mode-line-frame-identification
   "%b--"
   ;; Note that this is evaluated while making the list.
   ;; It makes a mode line construct which is just a string.
   (getenv "HOST")
   ":"
   'default-directory
   "   "
   'global-mode-string
   "   %[("
   '(:eval (format-time-string "%F"))
   'mode-line-process
   'minor-mode-alist
   "%n"
   ")%]--"
   '(which-function-mode ("" which-func-format "--"))
   '(line-number-mode "L%l--")
   '(column-number-mode "C%c--")
   '(-3 "%p")))
```

(The variables `line-number-mode`, `column-number-mode` and `which-function-mode` enable particular minor modes; as usual, these variable names are also the minor mode command names.)

24.4.4 Variables Used in the Mode Line

This section describes variables incorporated by the standard value of `mode-line-format` into the text of the mode line. There is nothing inherently special about these variables; any other variables could have the same effects on the mode line if the value of `mode-line-format` is changed to use them. However, various parts of Emacs set these variables on the understanding that they will control parts of the mode line; therefore, practically speaking, it is essential for the mode line to use them. Also see Section "Optional Mode Line" in *The GNU Emacs Manual*.

`mode-line-mule-info` [Variable]
: This variable holds the value of the mode line construct that displays information about the language environment, buffer coding system, and current input method. See Chapter 34 [Non-ASCII Characters], page 966.

`mode-line-modified` [Variable]
: This variable holds the value of the mode line construct that displays whether the current buffer is modified. Its default value displays '`**`' if the buffer is modified, '`--`' if the buffer is not modified, '`%%`' if the buffer is read only, and '`%*`' if the buffer is read only and modified.

  Changing this variable does not force an update of the mode line.

`mode-line-frame-identification` [Variable]
: This variable identifies the current frame. Its default value displays `" "` if you are using a window system which can show multiple frames, or `"-%F "` on an ordinary terminal which shows only one frame at a time.

`mode-line-buffer-identification` [Variable]
: This variable identifies the buffer being displayed in the window. Its default value displays the buffer name, padded with spaces to at least 12 columns.

`mode-line-position` [Variable]
: This variable indicates the position in the buffer. Its default value displays the buffer percentage and, optionally, the buffer size, the line number and the column number.

`mode-line-percent-position` [User Option]
: This option is used in `mode-line-position`. Its value specifies both the buffer percentage to display (one of `nil`, `"%o"`, `"%p"`, `"%P"` or `"%q"`, see Section 24.4.5 [%-Constructs], page 557) and a width to space-fill or truncate to. You are recommended to set this option with the `customize-variable` facility.

`vc-mode` [Variable]
: The variable `vc-mode`, buffer-local in each buffer, records whether the buffer's visited file is maintained with version control, and, if so, which kind. Its value is a string that appears in the mode line, or `nil` for no version control.

`mode-line-modes` [Variable]
: This variable displays the buffer's major and minor modes. Its default value also displays the recursive editing level, information on the process status, and whether narrowing is in effect.

`mode-line-remote` [Variable]
: This variable is used to show whether `default-directory` for the current buffer is remote.

`mode-line-client` [Variable]
: This variable is used to identify `emacsclient` frames.

The following three variables are used in `mode-line-modes`:

`mode-name` [Variable]
: This buffer-local variable holds the "pretty" name of the current buffer's major mode. Each major mode should set this variable so that the mode name will appear in the mode line. The value does not have to be a string, but can use any of the data types valid in a mode-line construct (see Section 24.4.2 [Mode Line Data], page 552). To compute the string that will identify the mode name in the mode line, use `format-mode-line` (see Section 24.4.8 [Emulating Mode Line], page 561).

`mode-line-process` [Variable]
: This buffer-local variable contains the mode line information on process status in modes used for communicating with subprocesses. It is displayed immediately following the major mode name, with no intervening space. For example, its value in the `*shell*` buffer is `(":%s")`, which allows the shell to display its status along with the major mode as: '`(Shell:run)`'. Normally this variable is `nil`.

`mode-line-front-space` [Variable]
: This variable is displayed at the front of the mode line. By default, this construct is displayed right at the beginning of the mode line, except that if there is a memory-full message, it is displayed first.

`mode-line-end-spaces` [Variable]
: This variable is displayed at the end of the mode line.

`mode-line-misc-info` [Variable]
: Mode line construct for miscellaneous information. By default, this shows the information specified by `global-mode-string`.

`mode-line-position-line-format` [Variable]
: The format used to display line numbers when `line-number-mode` (see Section "Optional Mode Line" in *The GNU Emacs Manual*) is switched on. '`%l`' in the format will be replaced with the line number.

`mode-line-position-column-format` [Variable]
: The format used to display column numbers when `column-number-mode` (see Section "Optional Mode Line" in *The GNU Emacs Manual*) is switched on. '`%c`' in the format will be replaced with a zero-based column number, and '`%C`' will be replaced with a one-based column number.

`mode-line-position-column-line-format` [Variable]
: The format used to display column numbers when both `line-number-mode` and `column-number-mode` are switched on. See the previous two variables for the meaning of the '`%l`', '`%c`' and '`%C`' format specs.

`minor-mode-alist` [Variable]

This variable holds an association list whose elements specify how the mode line should indicate that a minor mode is active. Each element of the `minor-mode-alist` should be a two-element list:

```
(minor-mode-variable mode-line-string)
```

More generally, *mode-line-string* can be any mode line construct. It appears in the mode line when the value of *minor-mode-variable* is non-`nil`, and not otherwise. These strings should begin with spaces so that they don't run together. Conventionally, the *minor-mode-variable* for a specific mode is set to a non-`nil` value when that minor mode is activated.

`minor-mode-alist` itself is not buffer-local. Each variable mentioned in the alist should be buffer-local if its minor mode can be enabled separately in each buffer.

`global-mode-string` [Variable]

This variable holds a mode line construct that, by default, appears in the mode line as part of `mode-line-misc-info`, just after the `which-function-mode` information if that minor mode is enabled, else after `mode-line-modes`. Elements that are added to this construct should normally end in a space (to ensure that consecutive `global-mode-string` elements display properly). For instance, the command `display-time` sets `global-mode-string` to refer to the variable `display-time-string`, which holds a string containing the time and load information.

The '`%M`' construct substitutes the value of `global-mode-string`. This construct is not used by the default mode line, as the variable itself is used in `mode-line-misc-info`.

Here is a simplified version of the default value of `mode-line-format`. The real default value also specifies addition of text properties.

```
("-"
 mode-line-mule-info
 mode-line-modified
 mode-line-frame-identification
 mode-line-buffer-identification
 "   "
 mode-line-position
 (vc-mode vc-mode)
 "   "
 mode-line-modes
 (which-function-mode ("" which-func-format "--"))
 (global-mode-string ("--" global-mode-string))
 "-%-")
```

24.4.5 %-Constructs in the Mode Line

Strings used as mode line constructs can use certain %-constructs to substitute various kinds of data. The following is a list of the defined %-constructs, and what they mean.

In any construct except '`%%`', you can add a decimal integer after the '`%`' to specify a minimum field width. If the width is less, the field is padded to that width. Purely numeric

constructs (‘`c`’, ‘`i`’, ‘`I`’, and ‘`l`’) are padded by inserting spaces to the left, and others are padded by inserting spaces to the right.

`%b` The current buffer name, obtained with the `buffer-name` function. See Section 28.3 [Buffer Names], page 676.

`%c` The current column number of point, counting from zero starting at the left margin of the window.

`%C` The current column number of point, counting from one starting at the left margin of the window.

`%e` When Emacs is nearly out of memory for Lisp objects, a brief message saying so. Otherwise, this is empty.

`%f` The visited file name, obtained with the `buffer-file-name` function. See Section 28.4 [Buffer File Name], page 677.

`%F` The title (only on a window system) or the name of the selected frame. See Section 30.4.3.1 [Basic Parameters], page 807.

`%i` The size of the accessible part of the current buffer; basically `(- (point-max) (point-min))`.

`%I` Like ‘`%i`’, but the size is printed in a more readable way by using ‘`k`’ for 10^3, ‘`M`’ for 10^6, ‘`G`’ for 10^9, etc., to abbreviate.

`%l` The current line number of point, counting within the accessible portion of the buffer.

`%M` The value of `global-mode-string` (which is part of `mode-line-misc-info` by default).

`%n` ‘`Narrow`’ when narrowing is in effect; nothing otherwise (see `narrow-to-region` in Section 31.4 [Narrowing], page 867).

`%o` The degree of *travel* of the window through (the visible portion of) the buffer, i.e. the size of the text above the top of the window expressed as a percentage of all the text outside the window, or ‘`Top`’, ‘`Bottom`’ or ‘`All`’.

`%p` The percentage of the buffer text above the **top** of window, or ‘`Top`’, ‘`Bottom`’ or ‘`All`’. Note that the default mode line construct truncates this to three characters.

`%P` The percentage of the buffer text that is above the **bottom** of the window (which includes the text visible in the window, as well as the text above the top), plus ‘`Top`’ if the top of the buffer is visible on screen; or ‘`Bottom`’ or ‘`All`’.

`%q` The percentages of text above both the **top** and the **bottom** of the window, separated by ‘`-`’, or ‘`All`’.

`%s` The status of the subprocess belonging to the current buffer, obtained with `process-status`. See Section 40.6 [Process Information], page 1095.

`%z` The mnemonics of keyboard, terminal, and buffer coding systems.

`%Z` Like ‘`%z`’, but including the end-of-line format.

`%*` '`%`' if the buffer is read only (see `buffer-read-only`);
'`*`' if the buffer is modified (see `buffer-modified-p`);
'`-`' otherwise. See Section 28.5 [Buffer Modification], page 679.

`%+` '`*`' if the buffer is modified (see `buffer-modified-p`);
'`%`' if the buffer is read only (see `buffer-read-only`);
'`-`' otherwise. This differs from '`%*`' only for a modified read-only buffer. See Section 28.5 [Buffer Modification], page 679.

`%&` '`*`' if the buffer is modified, and '`-`' otherwise.

`%@` '`@`' if the buffer's `default-directory` (see Section 26.9.4 [File Name Expansion], page 641) is on a remote machine, and '`-`' otherwise.

`%[` An indication of the depth of recursive editing levels (not counting minibuffer levels): one '`[`' for each editing level. See Section 22.13 [Recursive Editing], page 476.

`%]` One '`]`' for each recursive editing level (not counting minibuffer levels).

`%-` Dashes sufficient to fill the remainder of the mode line.

`%%` The character '`%`'—this is how to include a literal '`%`' in a string in which `%`-constructs are allowed.

Obsolete %-Constructs

The following constructs should no longer be used.

`%m` Obsolete; use the `mode-name` variable instead. The `%m` construct is inadequate, as it produces an empty string if the value of `mode-name` is a non-string mode-line construct (as in `emacs-lisp-mode`, for example).

24.4.6 Properties in the Mode Line

Certain text properties are meaningful in the mode line. The `face` property affects the appearance of text; the `help-echo` property associates help strings with the text, and `keymap` can make the text mouse-sensitive.

There are four ways to specify text properties for text in the mode line:

1. Put a string with a text property directly into the mode line data structure, but see Section 24.4.2 [Mode Line Data], page 552, for caveats for that.
2. Put a text property on a mode line %-construct such as '`%12b`'; then the expansion of the %-construct will have that same text property.
3. Use a `(:propertize` *`elt props...`*`)` construct to give *elt* a text property specified by *props*.
4. Use a list containing `:eval` *`form`* in the mode line data structure, and make *form* evaluate to a string that has a text property.

You can use the `keymap` property to specify a keymap. This keymap only takes real effect for mouse clicks; binding character keys and function keys to it has no effect, since it is impossible to move point into the mode line.

When the mode line refers to a variable which does not have a non-`nil` `risky-local-variable` property, any text properties given or specified within that variable's values are

ignored. This is because such properties could otherwise specify functions to be called, and those functions could come from file local variables.

24.4.7 Window Header Lines

A window can have a *header line* at the top, just as it can have a mode line at the bottom. The header line feature works just like the mode line feature, except that it's controlled by `header-line-format`:

`header-line-format` [Variable]

This variable, local in every buffer, specifies how to display the header line, for windows displaying the buffer. The format of the value is the same as for `mode-line-format` (see Section 24.4.2 [Mode Line Data], page 552). It is normally `nil`, so that ordinary buffers have no header line.

If `display-line-numbers-mode` is turned on in a buffer (see Section "Display Custom" in *The GNU Emacs Manual*), the buffer text is indented on display by the amount of screen space needed to show the line numbers. By contrast, text of the header line is not automatically indented, because a header line never displays a line number, and because the text of the header line is not necessarily directly related to buffer text below it. If a Lisp program needs the header-line text to be aligned with buffer text (for example, if the buffer displays columnar data, like `tabulated-list-mode` does, see Section 24.2.7 [Tabulated List Mode], page 539), it should turn on the minor mode `header-line-indent-mode`.

`header-line-indent-mode` [Command]

This buffer-local minor mode tracks the changes of the width of the line-number display on screen (which may vary depending on the range of line numbers shown in the window), and allows Lisp programs to arrange that header-line text is always aligned with buffer text when the line-number width changes. Such Lisp programs should turn on this mode in the buffer, and use the variables `header-line-indent` and `header-line-indent-width` in the `header-line-format` to ensure it is adjusted to the text indentation at all times.

`header-line-indent` [Variable]

This variable's value is a whitespace string whose width is kept equal to the current width of line-numbers on display, provided that `header-line-indent-mode` is turned on in the buffer shown in the window. The number of spaces is calculated under the assumption that the face of the header-line text uses the same font, including size, as the frame's default font; if that assumption is false, use `header-line-indent-width`, described below, instead. This variable is intended to be used in simple situations where the header-line text needs to be indented as a whole to be realigned with buffer text, by prepending this variable's value to the actual header-line text. For example, the following definition of `header-line-format`:

```
(setq header-line-format
      `("" header-line-indent ,my-header-line))
```

where `my-header-line` is the format string that produces the actual text of the header line, will make sure the header-line text is always indented like the buffer text below it.

`header-line-indent-width` [Variable]
: This variable's value is kept updated to provide the current width, in units of the frame's canonical character width, used for displaying the line numbers, provided that `header-line-indent-mode` is turned on in the buffer shown in the window. It can be used for aligning the header-line text with the buffer text when `header-line-indent` is not flexible enough. For example, if the header line uses a font whose metrics is different from the default face's font, your Lisp program can calculate the width of line-number display in pixels, by multiplying the value of this variable by the value returned by `frame-char-width` (see Section 30.3.2 [Frame Font], page 799), and then use the result to align header-line text using the `:align-to` display property spec (see Section 41.16.2 [Specified Space], page 1205) in pixels on the relevant parts of `header-line-format`.

`window-header-line-height` **&optional** *window* [Function]
: This function returns the height in pixels of *window*'s header line. *window* must be a live window, and defaults to the selected window.

A window that is just one line tall never displays a header line. A window that is two lines tall cannot display both a mode line and a header line at once; if it has a mode line, then it does not display a header line.

24.4.8 Emulating Mode Line Formatting

You can use the function `format-mode-line` to compute the text that would appear in a mode line or header line based on a certain mode line construct.

`format-mode-line` *format* **&optional** *face window buffer* [Function]
: This function formats a line of text according to *format* as if it were generating the mode line for *window*, but it also returns the text as a string. The argument *window* defaults to the selected window. If *buffer* is non-`nil`, all the information used is taken from *buffer*; by default, it comes from *window*'s buffer.

  The value string normally has text properties that correspond to the faces, keymaps, etc., that the mode line would have. Any character for which no `face` property is specified by *format* gets a default value determined by *face*. If *face* is `t`, that stands for either `mode-line` if *window* is selected, otherwise `mode-line-inactive`. If *face* is `nil` or omitted, that stands for the default face. If *face* is an integer, the value returned by this function will have no text properties.

  You can also specify other valid faces as the value of *face*. If specified, that face provides the `face` property for characters whose face is not specified by *format*.

  Note that using `mode-line`, `mode-line-inactive`, or `header-line` as *face* will actually redisplay the mode line or the header line, respectively, using the current definitions of the corresponding face, in addition to returning the formatted string. (Other faces do not cause redisplay.)

  For example, `(format-mode-line header-line-format)` returns the text that would appear in the selected window's header line (`""` if it has no header line). `(format-mode-line header-line-format 'header-line)` returns the same text, with each character carrying the face that it will have in the header line itself, and also redraws the header line.

24.5 Imenu

Imenu is a feature that lets users select a definition or section in the buffer, from a menu which lists all of them, to go directly to that location in the buffer. Imenu works by constructing a buffer index which lists the names and buffer positions of the definitions, or other named portions of the buffer; then the user can choose one of them and move point to it. Major modes can add a menu bar item to use Imenu using `imenu-add-to-menubar`.

`imenu-add-to-menubar` *name* [Command]
: This function defines a local menu bar item named *name* to run Imenu.

The user-level commands for using Imenu are described in the Emacs Manual (see Section "Imenu" in *the Emacs Manual*). This section explains how to customize Imenu's method of finding definitions or buffer portions for a particular major mode.

The usual and simplest way is to set the variable `imenu-generic-expression`:

`imenu-generic-expression` [Variable]
: This variable, if non-`nil`, is a list that specifies regular expressions for finding definitions for Imenu. Simple elements of `imenu-generic-expression` look like this:

  ```
  (menu-title regexp index)
  ```

  Here, if *menu-title* is non-`nil`, it says that the matches for this element should go in a submenu of the buffer index; *menu-title* itself specifies the name for the submenu. If *menu-title* is `nil`, the matches for this element go directly in the top level of the buffer index.

  The second item in the list, *regexp*, is a regular expression (see Section 35.3 [Regular Expressions], page 999); anything in the buffer that it matches is considered a definition, something to mention in the buffer index. The third item, *index*, is a non-negative integer that indicates which subexpression in *regexp* matches the definition's name.

  An element can also look like this:

  ```
  (menu-title regexp index function arguments...)
  ```

  Each match for this element creates an index item, and when the index item is selected by the user, it calls *function* with arguments consisting of the item name, the buffer position, and *arguments*.

  For Emacs Lisp mode, `imenu-generic-expression` could look like this:

  ```
  ((nil "^\\s-*(def\\(un\\|subst\\|macro\\|advice\\)\
  \\s-+\\([-A-Za-z0-9+]+\\)" 2)
   ("*Vars*" "^\\s-*(def\\(var\\|const\\)\
  \\s-+\\([-A-Za-z0-9+]+\\)" 2)
   ("*Types*"
    "^\\s-*\
  (def\\(type\\|struct\\|class\\|ine-condition\\)\
  \\s-+\\([-A-Za-z0-9+]+\\)" 2))
  ```

  Setting this variable makes it buffer-local in the current buffer.

`imenu-case-fold-search` [Variable]

This variable controls whether matching against the regular expressions in the value of `imenu-generic-expression` is case-sensitive: `t`, the default, means matching should ignore case.

Setting this variable makes it buffer-local in the current buffer.

`imenu-syntax-alist` [Variable]

This variable is an alist of syntax table modifiers to use while processing `imenu-generic-expression`, to override the syntax table of the current buffer. Each element should have this form:

```
(characters . syntax-description)
```

The CAR, *characters*, can be either a character or a string. The element says to give that character or characters the syntax specified by *syntax-description*, which is passed to `modify-syntax-entry` (see Section 36.3 [Syntax Table Functions], page 1030).

This feature is typically used to give word syntax to characters which normally have symbol syntax, and thus to simplify `imenu-generic-expression` and speed up matching. For example, Fortran mode uses it this way:

```
(setq imenu-syntax-alist '(("_$" . "w")))
```

The `imenu-generic-expression` regular expressions can then use '`\\sw+`' instead of '`\\(\\sw\\|\\s_\\)+`'. Note that this technique may be inconvenient when the mode needs to limit the initial character of a name to a smaller set of characters than are allowed in the rest of a name.

Setting this variable makes it buffer-local in the current buffer.

Another way to customize Imenu for a major mode is to set the variables `imenu-prev-index-position-function` and `imenu-extract-index-name-function`:

`imenu-prev-index-position-function` [Variable]

If this variable is non-`nil`, its value should be a function that finds the next definition to put in the buffer index, scanning backward in the buffer from point. It should return `nil` if it doesn't find another definition before point. Otherwise it should leave point at the place it finds a definition and return any non-`nil` value.

Setting this variable makes it buffer-local in the current buffer.

`imenu-extract-index-name-function` [Variable]

If this variable is non-`nil`, its value should be a function to return the name for a definition, assuming point is in that definition as the `imenu-prev-index-position-function` function would leave it.

Setting this variable makes it buffer-local in the current buffer.

The last way to customize Imenu for a major mode is to set the variable `imenu-create-index-function`:

`imenu-create-index-function` [Variable]

This variable specifies the function to use for creating a buffer index. The function should take no arguments, and return an index alist for the current buffer. It is called within `save-excursion`, so where it leaves point makes no difference.

The index alist can have three types of elements. Simple elements look like this:

```
(index-name . index-position)
```

Selecting a simple element has the effect of moving to position *index-position* in the buffer. Special elements look like this:

```
(index-name index-position function arguments...)
```

Selecting a special element performs:

```
(funcall function
         index-name index-position arguments...)
```

A nested sub-alist element looks like this:

```
(menu-title . sub-alist)
```

It creates the submenu *menu-title* specified by *sub-alist*.

The default value of `imenu-create-index-function` is `imenu-default-create-index-function`. This function calls the value of `imenu-prev-index-position-function` and the value of `imenu-extract-index-name-function` to produce the index alist. However, if either of these two variables is `nil`, the default function uses `imenu-generic-expression` instead.

Setting this variable makes it buffer-local in the current buffer.

If built with tree-sitter, Emacs can automatically generate an Imenu index if the major mode sets relevant variables.

`treesit-simple-imenu-settings` [Variable]

This variable instructs Emacs how to generate Imenu indexes. It should be a list of (*category regexp pred name-fn*).

category should be the name of a category, like `"Function"`, `"Class"`, etc. *regexp* should be a regexp matching the type of nodes that belong to *category*. *pred* should be either `nil` or a function that takes a node as the argument. It should return non-`nil` if the node is a valid node for *category*, or `nil` if not.

category could also be `nil`, in which case the entries matched by *regexp* and *pred* are not grouped under *category*.

name-fn should be either `nil` or a function that takes a defun node and returns the name of that defun, e.g., the function name for a function definition. If *name-fn* is `nil`, `treesit-defun-name` (see Section 37.7 [Tree-sitter Major Modes], page 1064) is used instead.

`treesit-major-mode-setup` (see Section 37.7 [Tree-sitter Major Modes], page 1064) automatically sets up Imenu if this variable is non-`nil`.

24.6 Font Lock Mode

Font Lock mode is a buffer-local minor mode that automatically attaches `face` properties to certain parts of the buffer based on their syntactic role. How it parses the buffer depends on the major mode; most major modes define syntactic criteria for which faces to use in which contexts. This section explains how to customize Font Lock for a particular major mode.

Font Lock mode finds text to highlight in three ways: through parsing based on a full-blown parser (usually, via an external library or program), through syntactic parsing based on the Emacs's built-in syntax table, or through searching (usually for regular expressions). If enabled, parser-based fontification happens first (see Section 24.6.10 [Parser-based Font Lock], page 580). Syntactic fontification happens next; it finds comments and string constants and highlights them. Search-based fontification happens last.

24.6.1 Font Lock Basics

The Font Lock functionality is based on several basic functions. Each of these calls the function specified by the corresponding variable. This indirection allows major and minor modes to modify the way fontification works in the buffers of that mode, and even use the Font Lock mechanisms for features that have nothing to do with fontification. (This is why the description below says "should" when it describes what the functions do: the mode can customize the values of the corresponding variables to do something entirely different.) The variables mentioned below are described in Section 24.6.4 [Other Font Lock Variables], page 572.

`font-lock-fontify-buffer`
: This function should fontify the current buffer's accessible portion, by calling the function specified by `font-lock-fontify-buffer-function`.

`font-lock-unfontify-buffer`
: Used when turning Font Lock off to remove the fontification. Calls the function specified by `font-lock-unfontify-buffer-function`.

`font-lock-fontify-region beg end &optional loudly`
: Should fontify the region between *beg* and *end*. If *loudly* is non-`nil`, should display status messages while fontifying. Calls the function specified by `font-lock-fontify-region-function`.

`font-lock-unfontify-region beg end`
: Should remove fontification from the region between *beg* and *end*. Calls the function specified by `font-lock-unfontify-region-function`.

`font-lock-flush &optional beg end`
: This function should mark the fontification of the region between *beg* and *end* as outdated. If not specified or `nil`, *beg* and *end* default to the beginning and end of the buffer's accessible portion. Calls the function specified by `font-lock-flush-function`.

`font-lock-ensure &optional beg end`
: This function should make sure the region between *beg* and *end* has been fontified. The optional arguments *beg* and *end* default to the beginning and the end of the buffer's accessible portion. Calls the function specified by `font-lock-ensure-function`.

`font-lock-debug-fontify`
: This is a convenience command meant to be used when developing font locking for a mode, and should not be called from Lisp code. It recomputes all the relevant variables and then calls `font-lock-fontify-region` on the entire buffer.

There are several variables that control how Font Lock mode highlights text. But major modes should not set any of these variables directly. Instead, they should set `font-lock-defaults` as a buffer-local variable. The value assigned to this variable is used, if and when Font Lock mode is enabled, to set all the other variables.

`font-lock-defaults` [Variable]

This variable is set by modes to specify how to fontify text in that mode. It automatically becomes buffer-local when set. If its value is `nil`, Font Lock mode does no highlighting, and you can use the '`Faces`' menu (under '`Edit`' and then '`Text Properties`' in the menu bar) to assign faces explicitly to text in the buffer.

If non-`nil`, the value should look like this:

```
(keywords [keywords-only [case-fold
 [syntax-alist other-vars...]]])
```

The first element, *keywords*, indirectly specifies the value of `font-lock-keywords` which directs search-based fontification. It can be a symbol, a variable or a function whose value is the list to use for `font-lock-keywords`. It can also be a list of several such symbols, one for each possible level of fontification. The first symbol specifies the '`mode default`' level of fontification, the next symbol level 1 fontification, the next level 2, and so on. The '`mode default`' level is normally the same as level 1. It is used when `font-lock-maximum-decoration` has a `nil` value. See Section 24.6.5 [Levels of Font Lock], page 574.

The second element, *keywords-only*, specifies the value of the variable `font-lock-keywords-only`. If this is omitted or `nil`, syntactic fontification (of strings and comments) is also performed. If this is non-`nil`, syntactic fontification is not performed. See Section 24.6.8 [Syntactic Font Lock], page 577.

The third element, *case-fold*, specifies the value of `font-lock-keywords-case-fold-search`. If it is non-`nil`, Font Lock mode ignores case during search-based fontification.

If the fourth element, *syntax-alist*, is non-`nil`, it should be a list of cons cells of the form `(`*`char-or-string`*` . `*`string`*`)`. These are used to set up a syntax table for syntactic fontification; the resulting syntax table is stored in `font-lock-syntax-table`. If *syntax-alist* is omitted or `nil`, syntactic fontification uses the syntax table returned by the `syntax-table` function. See Section 36.3 [Syntax Table Functions], page 1030.

All the remaining elements (if any) are collectively called *other-vars*. Each of these elements should have the form `(`*`variable`*` . `*`value`*`)`—which means, make *variable* buffer-local and then set it to *value*. You can use these *other-vars* to set other variables that affect fontification, aside from those you can control with the first five elements. See Section 24.6.4 [Other Font Lock Variables], page 572.

If your mode fontifies text explicitly by adding `font-lock-face` properties, it can specify `(nil t)` for `font-lock-defaults` to turn off all automatic fontification. However, this is not required; it is possible to fontify some things using `font-lock-face` properties and set up automatic fontification for other parts of the text.

24.6.2 Search-based Fontification

The variable which directly controls search-based fontification is `font-lock-keywords`, which is typically specified via the *keywords* element in `font-lock-defaults`.

`font-lock-keywords` [Variable]
: The value of this variable is a list of the keywords to highlight. Lisp programs should not set this variable directly. Normally, the value is automatically set by Font Lock mode, using the *keywords* element in `font-lock-defaults`. The value can also be altered using the functions `font-lock-add-keywords` and `font-lock-remove-keywords` (see Section 24.6.3 [Customizing Keywords], page 570).

Each element of `font-lock-keywords` specifies how to find certain cases of text, and how to highlight those cases. Font Lock mode processes the elements of `font-lock-keywords` one by one, and for each element, it finds and handles all matches. Ordinarily, once part of the text has been fontified already, this cannot be overridden by a subsequent match in the same text; but you can specify different behavior using the *override* element of a *subexp-highlighter*.

Each element of `font-lock-keywords` should have one of these forms:

`regexp`
: Highlight all matches for *regexp* using `font-lock-keyword-face`. For example,

```
;; Highlight occurrences of the word 'foo'
;; using font-lock-keyword-face.
"\\<foo\\>"
```

Be careful when composing these regular expressions; a poorly written pattern can dramatically slow things down! The function `regexp-opt` (see Section 35.3.3 [Regexp Functions], page 1008) is useful for calculating optimal regular expressions to match several keywords.

`function`
: Find text by calling *function*, and highlight the matches it finds using `font-lock-keyword-face`.

When *function* is called, it receives one argument, the limit of the search; it should begin searching at point, and not search beyond the limit. It should return non-`nil` if it succeeds, and set the match data to describe the match that was found. Returning `nil` indicates failure of the search.

Fontification will call *function* repeatedly with the same limit, and with point where the previous invocation left it, until *function* fails. On failure, *function* need not reset point in any particular way.

`(`*`matcher . subexp`*`)`
: In this kind of element, *matcher* is either a regular expression or a function, as described above. The CDR, *subexp*, specifies which subexpression of *matcher* should be highlighted (instead of the entire text that *matcher* matched).

```
;; Highlight the 'bar' in each occurrence of 'fubar',
;; using font-lock-keyword-face.
("fu\\(bar\\)" . 1)
```

`(`*`matcher . facespec`*`)`
: In this kind of element, *facespec* is an expression whose value specifies the face to use for highlighting. In the simplest case, *facespec* is a Lisp variable (a symbol) whose value is a face name.

```
;; Highlight occurrences of 'fubar',
;; using the face which is the value of fubar-face.
("fubar" . fubar-face)
```

However, *facespec* can also evaluate to a list of this form:

```
(subexp
(face face prop1 val1 prop2 val2...))
```

to specify the face *face* and various additional text properties to put on the text that matches. If you do this, be sure to add the other text property names that you set in this way to the value of `font-lock-extra-managed-props` so that the properties will also be cleared out when they are no longer appropriate. Alternatively, you can set the variable `font-lock-unfontify-region-function` to a function that clears these properties. See Section 24.6.4 [Other Font Lock Variables], page 572.

`(`*`matcher . subexp-highlighter`*`)`
: In this kind of element, *subexp-highlighter* is a list which specifies how to highlight matches found by *matcher*. It has the form:

```
(subexp facespec [override [laxmatch]])
```

The CAR, *subexp*, is an integer specifying which subexpression of the match to fontify (0 means the entire matching text). The second subelement, *facespec*, is an expression whose value specifies the face, as described above.

The last two values in *subexp-highlighter*, *override* and *laxmatch*, are optional flags. If *override* is `t`, this element can override existing fontification made by previous elements of `font-lock-keywords`. If it is `keep`, then each character is fontified if it has not been fontified already by some other element. If it is `prepend`, the face specified by *facespec* is added to the beginning of the `font-lock-face` property. If it is `append`, the face is added to the end of the `font-lock-face` property.

If *laxmatch* is non-`nil`, it means there should be no error if there is no subexpression numbered *subexp* in *matcher*. Obviously, fontification of the subexpression numbered *subexp* will not occur. However, fontification of other subexpressions (and other regexps) will continue. If *laxmatch* is `nil`, and the specified subexpression is missing, then an error is signaled which terminates search-based fontification.

Here are some examples of elements of this kind, and what they do:

```
;; Highlight occurrences of either 'foo' or 'bar', using
;; foo-bar-face, even if they have already been highlighted.
;; foo-bar-face should be a variable whose value is a face.
("foo\\|bar" 0 foo-bar-face t)

;; Highlight the first subexpression within each occurrence
;; that the function fubar-match finds,
;; using the face which is the value of fubar-face.
```

```
(fubar-match 1 fubar-face)
```

(*matcher* . *anchored-highlighter*)
: In this kind of element, *anchored-highlighter* specifies how to highlight text that follows a match found by *matcher*. So a match found by *matcher* acts as the anchor for further searches specified by *anchored-highlighter*. *anchored-highlighter* is a list of the following form:

  ```
  (anchored-matcher pre-form post-form subexp-highlighters...)
  ```

  Here, *anchored-matcher*, like *matcher*, is either a regular expression or a function. After a match of *matcher* is found, point is at the end of the match. Now, Font Lock evaluates the form *pre-form*. Then it searches for matches of *anchored-matcher* and uses *subexp-highlighters* to highlight these. A *subexp-highlighter* is as described above. Finally, Font Lock evaluates *post-form*.

  The forms *pre-form* and *post-form* can be used to initialize before, and cleanup after, *anchored-matcher* is used. Typically, *pre-form* is used to move point to some position relative to the match of *matcher*, before starting with *anchored-matcher*. *post-form* might be used to move back, before resuming with *matcher*.

  After Font Lock evaluates *pre-form*, it does not search for *anchored-matcher* beyond the end of the line. However, if *pre-form* returns a buffer position that is greater than the position of point after *pre-form* is evaluated, then the position returned by *pre-form* is used as the limit of the search instead. It is generally a bad idea to return a position greater than the end of the line; in other words, the *anchored-matcher* search should not span lines.

  For example,

  ```
  ;; Highlight occurrences of the word 'item' following
  ;; an occurrence of the word 'anchor' (on the same line)
  ;; in the value of item-face.
  ("\\<anchor\\>" "\\<item\\>" nil nil (0 item-face))
  ```

  Here, *pre-form* and *post-form* are `nil`. Therefore searching for '`item`' starts at the end of the match of '`anchor`', and searching for subsequent instances of '`anchor`' resumes from where searching for '`item`' concluded.

(*matcher highlighters...*)
: This sort of element specifies several *highlighter* lists for a single *matcher*. A *highlighter* list can be of the type *subexp-highlighter* or *anchored-highlighter* as described above.

  For example,

  ```
  ;; Highlight occurrences of the word 'anchor' in the value
  ;; of anchor-face, and subsequent occurrences of the word
  ;; 'item' (on the same line) in the value of item-face.
  ("\\<anchor\\>" (0 anchor-face)
                  ("\\<item\\>" nil nil (0 item-face)))
  ```

(eval . *form*)
: Here *form* is an expression to be evaluated the first time this value of `font-lock-keywords` is used in a buffer. Its value should have one of the forms described in this table.

Warning: Do not design an element of `font-lock-keywords` to match text which spans lines; this does not work reliably. For details, see Section 24.6.9 [Multiline Font Lock], page 577.

You can use *case-fold* in `font-lock-defaults` to specify the value of `font-lock-keywords-case-fold-search` which says whether search-based fontification should be case-insensitive.

`font-lock-keywords-case-fold-search` [Variable]
: Non-`nil` means that regular expression matching for the sake of `font-lock-keywords` should be case-insensitive.

24.6.3 Customizing Search-Based Fontification

You can use `font-lock-add-keywords` to add additional search-based fontification rules to a major mode, and `font-lock-remove-keywords` to remove rules. You can also customize the `font-lock-ignore` option to selectively disable fontification rules for keywords that match certain criteria.

`font-lock-add-keywords` *mode keywords* **&optional** *how* [Function]
: This function adds highlighting *keywords*, for the current buffer or for major mode *mode*. The argument *keywords* should be a list with the same format as the variable `font-lock-keywords`.

  If *mode* is a symbol which is a major mode command name, such as `c-mode`, the effect is that enabling Font Lock mode in *mode* will add *keywords* to `font-lock-keywords`. Calling with a non-`nil` value of *mode* is correct only in your `~/.emacs` file.

  If *mode* is `nil`, this function adds *keywords* to `font-lock-keywords` in the current buffer. This way of calling `font-lock-add-keywords` is usually used in mode hook functions.

  By default, *keywords* are added at the beginning of `font-lock-keywords`. If the optional argument *how* is `set`, they are used to replace the value of `font-lock-keywords`. If *how* is any other non-`nil` value, they are added at the end of `font-lock-keywords`.

  Some modes provide specialized support you can use in additional highlighting patterns. See the variables `c-font-lock-extra-types`, `c++-font-lock-extra-types`, and `java-font-lock-extra-types`, for example.

  Warning: Major mode commands must not call `font-lock-add-keywords` under any circumstances, either directly or indirectly, except through their mode hooks. (Doing so would lead to incorrect behavior for some minor modes.) They should set up their rules for search-based fontification by setting `font-lock-keywords`.

`font-lock-remove-keywords` *mode keywords* [Function]
: This function removes *keywords* from `font-lock-keywords` for the current buffer or for major mode *mode*. As in `font-lock-add-keywords`, *mode* should be a major mode command name or `nil`. All the caveats and requirements for `font-lock-add-keywords` apply here too. The argument *keywords* must exactly match the one used by the corresponding `font-lock-add-keywords`.

For example, the following code adds two fontification patterns for C mode: one to fontify the word 'FIXME', even in comments, and another to fontify the words 'and', 'or' and 'not' as keywords.

```
(font-lock-add-keywords 'c-mode
 '(("\\<\\(FIXME\\):" 1 font-lock-warning-face prepend)
   ("\\<\\(and\\|or\\|not\\)\\>" . font-lock-keyword-face)))
```

This example affects only C mode proper. To add the same patterns to C mode *and* all modes derived from it, do this instead:

```
(add-hook 'c-mode-hook
 (lambda ()
  (font-lock-add-keywords nil
   '(("\\<\\(FIXME\\):" 1 font-lock-warning-face prepend)
     ("\\<\\(and\\|or\\|not\\)\\>" .
      font-lock-keyword-face)))))
```

`font-lock-ignore` [User Option]

This option defines conditions for selectively disabling fontifications due to certain Font Lock keywords. If non-`nil`, its value is a list of elements of the following form:

```
(symbol condition ...)
```

Here, *symbol* is a symbol, usually a major or minor mode. The subsequent *condition*s of a *symbol*'s list element will be in effect if *symbol* is bound and its value is non-`nil`. For a mode's symbol, it means that the current major mode is derived from that mode, or that minor mode is enabled in the buffer. When a *condition* is in effect, any fontifications caused by `font-lock-keywords` elements that match the *condition* will be disabled.

Each *condition* can be one of the following:

a symbol
: This condition matches any element of Font Lock keywords that references the symbol. This is usually a face, but can be any symbol referenced by an element of the `font-lock-keywords` list. The symbol can contain wildcards: `*` matches any string in the symbol'ss name, `?` matches a single character, and `[`*char-set*`]`, where *char-set* is a string of one or more characters, matches a single character from the set.

a string
: This condition matches any element of Font Lock keywords whose *matcher* is a regexp which matches the string. In other words, this condition matches a Font Lock rule which highlights the string. Thus, the string could be a specific program keyword whose highlighting you want to disable.

`(pred` *function*`)`
: This condition matches any element of Font Lock keywords for which *function*, when called with the element as the argument, returns non-`nil`.

`(not` *condition*`)`
: This matches if *condition* doesn't.

`(and` *condition* `...)`
: This matches if each of the *condition*s matches.

`(or` *condition* `...)`
: This matches if at least one of the *condition*s matches.

`(except` *condition*`)`
: This condition can only be used at top level or inside an `or` clause. It undoes the effect of a previously matching condition on the same level.

As an example, consider the following setting:

```
(setq font-lock-ignore
      '((prog-mode font-lock-*-face
                   (except help-echo))
        (emacs-lisp-mode (except ";;;###autoload)")
        (whitespace-mode whitespace-empty-at-bob-regexp)
        (makefile-mode (except *))))
```

Line by line, this does the following:

1. In all programming modes, disable fontifications due to all font-lock keywords that apply one of the standard font-lock faces (excluding strings and comments, which are covered by syntactic Font Lock).
2. However, keep any keywords that add a `help-echo` text property.
3. In Emacs Lisp mode, also keep the highlighting of autoload cookies, which would have been excluded by the first condition.
4. When `whitespace-mode` (a minor mode) is enabled, also don't highlight an empty line at beginning of buffer.
5. Finally, in Makefile mode, don't apply any conditions.

24.6.4 Other Font Lock Variables

This section describes additional variables that a major mode can set by means of *other-vars* in `font-lock-defaults` (see Section 24.6.1 [Font Lock Basics], page 565).

`font-lock-mark-block-function` [Variable]
: If this variable is non-`nil`, it should be a function that is called with no arguments, to choose an enclosing range of text for refontification for the command *`M-x font-lock-fontify-block`*.

  The function should report its choice by placing the region around it. A good choice is a range of text large enough to give proper results, but not too large so that refontification becomes slow. Typical values are `mark-defun` for programming modes or `mark-paragraph` for textual modes.

`font-lock-extra-managed-props` [Variable]
: This variable specifies additional properties (other than `font-lock-face`) that are being managed by Font Lock mode. It is used by `font-lock-default-unfontify-region`, which normally only manages the `font-lock-face` property. If you want Font Lock to manage other properties as well, you must specify them in a *facespec* in `font-lock-keywords` as well as add them to this list. See Section 24.6.2 [Search-based Fontification], page 567.

`font-lock-fontify-buffer-function` [Variable]
: Function to use for fontifying the buffer. The default value is `font-lock-default-fontify-buffer`.

`font-lock-unfontify-buffer-function` [Variable]
Function to use for unfontifying the buffer. This is used when turning off Font Lock mode. The default value is `font-lock-default-unfontify-buffer`.

`font-lock-fontify-region-function` [Variable]
Function to use for fontifying a region. It should take two arguments, the beginning and end of the region, and an optional third argument *verbose*. If *verbose* is non-`nil`, the function should print status messages. The default value is `font-lock-default-fontify-region`.

`font-lock-unfontify-region-function` [Variable]
Function to use for unfontifying a region. It should take two arguments, the beginning and end of the region. The default value is `font-lock-default-unfontify-region`.

`font-lock-flush-function` [Variable]
Function to use for declaring that a region's fontification is out of date. It takes two arguments, the beginning and end of the region. The default value of this variable is `font-lock-after-change-function`.

`font-lock-ensure-function` [Variable]
Function to use for making sure a region of the current buffer has been fontified. It is called with two arguments, the beginning and end of the region. The default value of this variable is a function that calls `font-lock-default-fontify-buffer` if the buffer is not fontified; the effect is to make sure the entire accessible portion of the buffer is fontified.

`jit-lock-register` *function* **&optional** *contextual* [Function]
This function tells Font Lock mode to run the Lisp function *function* any time it has to fontify or refontify part of the current buffer. It calls *function* before calling the default fontification functions, and gives it two arguments, *start* and *end*, which specify the region to be fontified or refontified. If *function* performs fontifications, it can return a list of the form `(jit-lock-bounds` *beg* `.` *end*`)`, to indicate the bounds of the region it actually fontified; Just-In-Time (a.k.a. "JIT") font-lock will use this information to optimize subsequent redisplay cycles and regions of buffer text it will pass to future calls to *function*.

The optional argument *contextual*, if non-`nil`, forces Font Lock mode to always refontify a syntactically relevant part of the buffer, and not just the modified lines. This argument can usually be omitted.

When Font Lock is activated in a buffer, it calls this function with a non-`nil` value of *contextual* if the value of `font-lock-keywords-only` (see Section 24.6.8 [Syntactic Font Lock], page 577) is `nil`.

`jit-lock-unregister` *function* [Function]
If *function* was previously registered as a fontification function using `jit-lock-register`, this function unregisters it.

`jit-lock-debug-mode` **&optional** *arg* [Command]
This is a minor mode whose purpose is to help in debugging code that is run by JIT font-lock. When this mode is enabled, most of the code that JIT font-lock

normally runs during redisplay cycles, where Lisp errors are suppressed, is instead run by a timer. Thus, this mode allows using debugging aids such as `debug-on-error` (see Section 19.1.1 [Error Debugging], page 334) and Edebug (see Section 19.2 [Edebug], page 345) for finding and fixing problems in font-lock code and any other code run by JIT font-lock. Another command that could be useful when developing and debugging font-lock is `font-lock-debug-fontify`, see Section 24.6.1 [Font Lock Basics], page 565.

24.6.5 Levels of Font Lock

Some major modes offer three different levels of fontification. You can define multiple levels by using a list of symbols for *keywords* in `font-lock-defaults`. Each symbol specifies one level of fontification; it is up to the user to choose one of these levels, normally by setting `font-lock-maximum-decoration` (see Section "Font Lock" in *the GNU Emacs Manual*). The chosen level's symbol value is used to initialize `font-lock-keywords`.

Here are the conventions for how to define the levels of fontification:

- Level 1: highlight function declarations, file directives (such as include or import directives), strings and comments. The idea is speed, so only the most important and top-level components are fontified.
- Level 2: in addition to level 1, highlight all language keywords, including type names that act like keywords, as well as named constant values. The idea is that all keywords (either syntactic or semantic) should be fontified appropriately.
- Level 3: in addition to level 2, highlight the symbols being defined in function and variable declarations, and all builtin function names, wherever they appear.

24.6.6 Precalculated Fontification

Some major modes such as `list-buffers` and `occur` construct the buffer text programmatically. The easiest way for them to support Font Lock mode is to specify the faces of text when they insert the text in the buffer.

The way to do this is to specify the faces in the text with the special text property `font-lock-face` (see Section 33.19.4 [Special Properties], page 927). When Font Lock mode is enabled, this property controls the display, just like the `face` property. When Font Lock mode is disabled, `font-lock-face` has no effect on the display.

It is ok for a mode to use `font-lock-face` for some text and also use the normal Font Lock machinery. But if the mode does not use the normal Font Lock machinery, it should not set the variable `font-lock-defaults`. In this case the `face` property will not be overridden, so using the `face` property could work too. However, using `font-lock-face` is generally preferable as it allows the user to control the fontification by toggling `font-lock-mode`, and lets the code work regardless of whether the mode uses Font Lock machinery or not.

24.6.7 Faces for Font Lock

Font Lock mode can highlight using any face, but Emacs defines several faces specifically for Font Lock to use to highlight text. These *Font Lock faces* are listed below. They can also be used by major modes for syntactic highlighting outside of Font Lock mode (see Section 24.2.1 [Major Mode Conventions], page 529).

Each of these symbols is both a face name, and a variable whose default value is the symbol itself. Thus, the default value of `font-lock-comment-face` is `font-lock-comment-face`.

The faces are listed with descriptions of their typical usage, and in order of greater to lesser prominence. If a mode's syntactic categories do not fit well with the usage descriptions, the faces can be assigned using the ordering as a guide.

`font-lock-warning-face`
: for a construct that is peculiar (e.g., an unescaped confusable quote in an Emacs Lisp symbol like '`‘foo`'), or that greatly changes the meaning of other text, like '`;;;###autoload`' in Emacs Lisp and '`#error`' in C.

`font-lock-function-name-face`
: for the name of a function being defined or declared.

`font-lock-function-call-face`
: for the name of a function being called. This face inherits, by default, from `font-lock-function-name-face`.

`font-lock-variable-name-face`
: for the name of a variable being defined or declared.

`font-lock-variable-use-face`
: for the name of a variable being referenced. This face inherits, by default, from `font-lock-variable-name-face`.

`font-lock-keyword-face`
: for a keyword with special syntactic significance, like '`for`' and '`if`' in C.

`font-lock-comment-face`
: for comments.

`font-lock-comment-delimiter-face`
: for comments delimiters, like '`/*`' and '`*/`' in C. On most terminals, this inherits from `font-lock-comment-face`.

`font-lock-type-face`
: for the names of user-defined data types.

`font-lock-constant-face`
: for the names of constants, like '`NULL`' in C.

`font-lock-builtin-face`
: for the names of built-in functions.

`font-lock-preprocessor-face`
: for preprocessor commands. This inherits, by default, from `font-lock-builtin-face`.

`font-lock-string-face`
: for string constants.

`font-lock-doc-face`
: for documentation embedded in program code inside specially-formed comments or strings. This face inherits, by default, from `font-lock-string-face`.

`font-lock-doc-markup-face`
: for mark-up elements in text using `font-lock-doc-face`. It is typically used for the mark-up constructs in documentation embedded in program code, following conventions such as Haddock, Javadoc or Doxygen. This face inherits, by default, from `font-lock-constant-face`.

`font-lock-negation-char-face`
: for easily-overlooked negation characters.

`font-lock-escape-face`
: for escape sequences in strings. This face inherits, by default, from `font-lock-regexp-grouping-backslash`.

  Here is an example in Python, where the escape sequence `\n` is used:

```
print('Hello world!\n')
```

`font-lock-number-face`
: for numbers.

`font-lock-operator-face`
: for operators.

`font-lock-property-name-face`
: for properties of an object, such as the declaration of fields in a struct. This face inherits, by default, from `font-lock-variable-name-face`.

`font-lock-property-use-face`
: for properties of an object, such as use of fields in a struct. This face inherits, by default, from `font-lock-property-name-face`.

  For example,

```
typedef struct
{
  int prop;
//    ^ property
} obj;

int main()
{
  obj o;
  o.prop = 3;
//  ^ property
}
```

`font-lock-punctuation-face`
: for punctuation such as brackets and delimiters.

`font-lock-bracket-face`
: for brackets (e.g., `()`, `[]`, `{}`). This face inherits, by default, from `font-lock-punctuation-face`.

`font-lock-delimiter-face`
: for delimiters (e.g., `;`, `:`, `,`). This face inherits, by default, from `font-lock-punctuation-face`.

`font-lock-misc-punctuation-face`
: for punctuation that is not a bracket or delimiter. This face inherits, by default, from `font-lock-punctuation-face`.

24.6.8 Syntactic Font Lock

Syntactic fontification uses a syntax table (see Chapter 36 [Syntax Tables], page 1025) to find and highlight syntactically relevant text. If enabled, it runs prior to search-based fontification. The variable `font-lock-syntactic-face-function`, documented below, determines which syntactic constructs to highlight. There are several variables that affect syntactic fontification; you should set them by means of `font-lock-defaults` (see Section 24.6.1 [Font Lock Basics], page 565).

Whenever Font Lock mode performs syntactic fontification on a stretch of text, it first calls the function specified by `syntax-propertize-function`. Major modes can use this to apply `syntax-table` text properties to override the buffer's syntax table in special cases. See Section 36.4 [Syntax Properties], page 1032.

`font-lock-keywords-only` [Variable]

If the value of this variable is non-`nil`, Font Lock does not do syntactic fontification, only search-based fontification based on `font-lock-keywords`. It is normally set by Font Lock mode based on the *keywords-only* element in `font-lock-defaults`. If the value is `nil`, Font Lock will call `jit-lock-register` (see Section 24.6.4 [Other Font Lock Variables], page 572) to set up for automatic refontification of buffer text following a modified line to reflect the new syntactic context due to the change.

To use only syntactic fontification, this variable should be non-`nil`, while `font-lock-keywords` should be set to `nil` (see Section 24.6.1 [Font Lock Basics], page 565).

`font-lock-syntax-table` [Variable]

This variable holds the syntax table to use for fontification of comments and strings. It is normally set by Font Lock mode based on the *syntax-alist* element in `font-lock-defaults`. If this value is `nil`, syntactic fontification uses the buffer's syntax table (the value returned by the function `syntax-table`; see Section 36.3 [Syntax Table Functions], page 1030).

`font-lock-syntactic-face-function` [Variable]

If this variable is non-`nil`, it should be a function to determine which face to use for a given syntactic element (a string or a comment).

The function is called with one argument, the parse state at point returned by `parse-partial-sexp`, and should return a face. The default value returns `font-lock-comment-face` for comments and `font-lock-string-face` for strings (see Section 24.6.7 [Faces for Font Lock], page 574).

This variable is normally set through the "other" elements in `font-lock-defaults`:

```
(setq-local font-lock-defaults
            `(,python-font-lock-keywords
              nil nil nil nil
              (font-lock-syntactic-face-function
               . python-font-lock-syntactic-face-function)))
```

24.6.9 Multiline Font Lock Constructs

Normally, elements of `font-lock-keywords` should not match across multiple lines; that doesn't work reliably, because Font Lock usually scans just part of the buffer, and it can

miss a multi-line construct that crosses the line boundary where the scan starts. (The scan normally starts at the beginning of a line.)

Making elements that match multiline constructs work properly has two aspects: correct *identification* and correct *rehighlighting*. The first means that Font Lock finds all multiline constructs. The second means that Font Lock will correctly rehighlight all the relevant text when a multiline construct is changed—for example, if some of the text that was previously part of a multiline construct ceases to be part of it. The two aspects are closely related, and often getting one of them to work will appear to make the other also work. However, for reliable results you must attend explicitly to both aspects.

There are three ways to ensure correct identification of multiline constructs:

- Add a function to `font-lock-extend-region-functions` that does the *identification* and extends the scan so that the scanned text never starts or ends in the middle of a multiline construct.
- Use the `font-lock-fontify-region-function` hook similarly to extend the scan so that the scanned text never starts or ends in the middle of a multiline construct.
- Somehow identify the multiline construct right when it gets inserted into the buffer (or at any point after that but before font-lock tries to highlight it), and mark it with a `font-lock-multiline` which will instruct font-lock not to start or end the scan in the middle of the construct.

There are several ways to do rehighlighting of multiline constructs:

- Place a `font-lock-multiline` property on the construct. This will rehighlight the whole construct if any part of it is changed. In some cases you can do this automatically by setting the `font-lock-multiline` variable, which see.
- Make sure `jit-lock-contextually` is set and rely on it doing its job. This will only rehighlight the part of the construct that follows the actual change, and will do it after a short delay. This only works if the highlighting of the various parts of your multiline construct never depends on text in subsequent lines. Since `jit-lock-contextually` is activated by default, this can be an attractive solution.
- Place a `jit-lock-defer-multiline` property on the construct. This works only if `jit-lock-contextually` is used, and with the same delay before rehighlighting, but like `font-lock-multiline`, it also handles the case where highlighting depends on subsequent lines.
- If parsing the *syntax* of a construct depends on it being parsed in one single chunk, you can add the `syntax-multiline` text property over the construct in question. The most common use for this is when the syntax property to apply to '`FOO`' depend on some later text '`BAR`': By placing this text property over the whole of '`FOO...BAR`', you make sure that any change of '`BAR`' will also cause the syntax property of '`FOO`' to be recomputed. Note: For this to work, the mode needs to add `syntax-propertize-multiline` to `syntax-propertize-extend-region-functions`.

24.6.9.1 Font Lock Multiline

One way to ensure reliable rehighlighting of multiline Font Lock constructs is to put on them the text property `font-lock-multiline`. It should be present and non-`nil` for text that is part of a multiline construct.

When Font Lock is about to highlight a range of text, it first extends the boundaries of the range as necessary so that they do not fall within text marked with the `font-lock-multiline` property. Then it removes any `font-lock-multiline` properties from the range, and highlights it. The highlighting specification (mostly `font-lock-keywords`) must reinstall this property each time, whenever it is appropriate.

Warning: don't use the `font-lock-multiline` property on large ranges of text, because that will make rehighlighting slow.

`font-lock-multiline` [Variable]

If the `font-lock-multiline` variable is set to `t`, Font Lock will try to add the `font-lock-multiline` property automatically on multiline constructs. This is not a universal solution, however, since it slows down Font Lock somewhat. It can miss some multiline constructs, or make the property larger or smaller than necessary.

For elements whose *matcher* is a function, the function should ensure that submatch 0 covers the whole relevant multiline construct, even if only a small subpart will be highlighted. It is often just as easy to add the `font-lock-multiline` property by hand.

The `font-lock-multiline` property is meant to ensure proper refontification; it does not automatically identify new multiline constructs. Identifying them requires that Font Lock mode operate on large enough chunks at a time. This will happen by accident on many cases, which may give the impression that multiline constructs magically work. If you set the `font-lock-multiline` variable non-`nil`, this impression will be even stronger, since the highlighting of those constructs which are found will be properly updated from then on. But that does not work reliably.

To find multiline constructs reliably, you must either manually place the `font-lock-multiline` property on the text before Font Lock mode looks at it, or use `font-lock-fontify-region-function`.

24.6.9.2 Region to Fontify after a Buffer Change

When a buffer is changed, the region that Font Lock refontifies is by default the smallest sequence of whole lines that spans the change. While this works well most of the time, sometimes it doesn't—for example, when a change alters the syntactic meaning of text on an earlier line.

You can enlarge (or even reduce) the region to refontify by setting the following variable:

`font-lock-extend-after-change-region-function` [Variable]

This buffer-local variable is either `nil` or a function for Font Lock mode to call to determine the region to scan and fontify.

The function is given three parameters, the standard *beg*, *end*, and *old-len* from `after-change-functions` (see Section 33.34 [Change Hooks], page 964). It should return either a cons of the beginning and end buffer positions (in that order) of the region to fontify, or `nil` (which means choose the region in the standard way). This function needs to preserve point, the match-data, and the current restriction. The region it returns may start or end in the middle of a line.

Since this function is called after every buffer change, it should be reasonably fast.

24.6.10 Parser-based Font Lock

Besides simple syntactic font lock and regexp-based font lock, Emacs also provides complete syntactic font lock with the help of a parser. Currently, Emacs uses the tree-sitter library (see Chapter 37 [Parsing Program Source], page 1042) for this purpose.

Parser-based font lock and other font lock mechanisms are not mutually exclusive. By default, if enabled, parser-based font lock runs first, replacing syntactic font lock, followed by regexp-based font lock.

Although parser-based font lock doesn't share the same customization variables with regexp-based font lock, it uses similar customization schemes. The tree-sitter counterpart of `font-lock-keywords` is `treesit-font-lock-settings`.

In general, tree-sitter fontification works as follows:

- A Lisp program (usually, part of a major mode) provides a *query* consisting of *patterns*, each pattern associated with a *capture name*.
- The tree-sitter library finds the nodes in the parse tree that match these patterns, tags the nodes with the corresponding capture names, and returns them to the Lisp program.
- The Lisp program uses the returned nodes to highlight the portions of buffer text corresponding to each node as appropriate, using the tagged capture names of the nodes to determine the correct fontification. For example, a node tagged `font-lock-keyword` would be highlighted in `font-lock-keyword` face.

For more information about queries, patterns, and capture names, see Section 37.5 [Pattern Matching], page 1055.

To set up tree-sitter fontification, a major mode should first set `treesit-font-lock-settings` with the output of `treesit-font-lock-rules`, then call `treesit-major-mode-setup`.

`treesit-font-lock-rules` **&rest** *query-specs* [Function]

This function is used to set `treesit-font-lock-settings`. It takes care of compiling queries and other post-processing, and outputs a value that `treesit-font-lock-settings` accepts. Here's an example:

```
(treesit-font-lock-rules
 :language 'javascript
 :feature 'constant
 :override t
 '((true) @font-lock-constant-face
   (false) @font-lock-constant-face)
 :language 'html
 :feature 'script
 "(script_element) @font-lock-builtin-face")
```

This function takes a series of *query-spec*s, where each *query-spec* is a *query* preceded by one or more *keyword*/*value* pairs. Each *query* is a tree-sitter query in either the string, s-expression, or compiled form.

For each *query*, the *keyword*/*value* pairs that precede it add meta information to it. The `:language` keyword declares *query*'s language. The `:feature` keyword

sets the feature name of *query*. Users can control which features are enabled with `treesit-font-lock-level` and `treesit-font-lock-feature-list` (described below). These two keywords are mandatory.

Other keywords are optional:

Keyword	Value	Description
`:override`	`nil`	If the region already has a face, discard the new face
	`t`	Always apply the new face
	`append`	Append the new face to existing ones
	`prepend`	Prepend the new face to existing ones
	`keep`	Fill-in regions without an existing face

Lisp programs mark patterns in *query* with capture names (names that start with `@`), and tree-sitter will return matched nodes tagged with those same capture names. For the purpose of fontification, capture names in *query* should be face names like `font-lock-keyword-face`. The captured node will be fontified with that face.

A capture name can also be a function name, in which case the function is called with 4 arguments: *node* and *override*, *start* and *end*, where *node* is the node itself, *override* is the `:override` property of the rule which captured this node, and *start* and *end* limit the region which this function should fontify. (If this function wants to respect the *override* argument, it can use `treesit-fontify-with-override`.)

Beyond the 4 arguments presented, this function should accept more arguments as optional arguments for future extensibility.

If a capture name is both a face and a function, the face takes priority. If a capture name is neither a face nor a function, it is ignored.

`treesit-font-lock-feature-list` [Variable]

This is a list of lists of feature symbols. Each element of the list is a list that represents a decoration level. `treesit-font-lock-level` controls which levels are activated.

Each element of the list is a list of the form `(`*`feature`* `...)`, where each *feature* corresponds to the `:feature` value of a query defined in `treesit-font-lock-rules`. Removing a feature symbol from this list disables the corresponding query during font-lock.

Common feature names, for many programming languages, include `definition`, `type`, `assignment`, `builtin`, `constant`, `keyword`, `string-interpolation`, `comment`, `doc`, `string`, `operator`, `preprocessor`, `escape-sequence`, and `key`. Major modes are free to subdivide or extend these common features.

Some of these features warrant some explanation: `definition` highlights whatever is being defined, e.g., the function name in a function definition, the struct name in a struct definition, the variable name in a variable definition; `assignment` highlights whatever is being assigned to, e.g., the variable or field in an assignment statement; `key` highlights keys in key-value pairs, e.g., keys in a JSON object or Python dictionary; `doc` highlights docstrings or doc-comments.

For example, the value of this variable could be:

```
((comment string doc) ; level 1
 (function-name keyword type builtin constant) ; level 2
 (variable-name string-interpolation key)) ; level 3
```

Major modes should set this variable before calling `treesit-major-mode-setup`.

For this variable to take effect, a Lisp program should call `treesit-font-lock-recompute-features` (which resets `treesit-font-lock-settings` accordingly), or `treesit-major-mode-setup` (which calls `treesit-font-lock-recompute-features`).

`treesit-font-lock-settings` [Variable]

A list of settings for tree-sitter based font lock. The exact format of each setting is considered internal. One should always use `treesit-font-lock-rules` to set this variable.

Multi-language major modes should provide range functions in `treesit-range-functions`, and Emacs will set the ranges accordingly before fontifing a region (see Section 37.6 [Multiple Languages], page 1060).

24.7 Automatic Indentation of code

For programming languages, an important feature of a major mode is to provide automatic indentation. There are two parts: one is to decide what is the right indentation of a line, and the other is to decide when to reindent a line. By default, Emacs reindents a line whenever you type a character in `electric-indent-chars`, which by default only includes Newline. Major modes can add chars to `electric-indent-chars` according to the syntax of the language.

Deciding what is the right indentation is controlled in Emacs by `indent-line-function` (see Section 33.17.2 [Mode-Specific Indent], page 913). For some modes, the *right* indentation cannot be known reliably, typically because indentation is significant so several indentations are valid but with different meanings. In that case, the mode should set `electric-indent-inhibit` to make sure the line is not constantly re-indented against the user's wishes.

Writing a good indentation function can be difficult and to a large extent it is still a black art. Many major mode authors will start by writing a simple indentation function that works for simple cases, for example by comparing with the indentation of the previous text line. For most programming languages that are not really line-based, this tends to scale very poorly: improving such a function to let it handle more diverse situations tends to become more and more difficult, resulting in the end with a large, complex, unmaintainable indentation function which nobody dares to touch.

A good indentation function will usually need to actually parse the text, according to the syntax of the language. Luckily, it is not necessary to parse the text in as much detail as would be needed for a compiler, but on the other hand, the parser embedded in the indentation code will want to be somewhat friendly to syntactically incorrect code.

Good maintainable indentation functions usually fall into two categories: either parsing forward from some safe starting point until the position of interest, or parsing backward from the position of interest. Neither of the two is a clearly better choice than the other: parsing

backward is often more difficult than parsing forward because programming languages are designed to be parsed forward, but for the purpose of indentation it has the advantage of not needing to guess a safe starting point, and it generally enjoys the property that only a minimum of text will be analyzed to decide the indentation of a line, so indentation will tend to be less affected by syntax errors in some earlier unrelated piece of code. Parsing forward on the other hand is usually easier and has the advantage of making it possible to reindent efficiently a whole region at a time, with a single parse.

Rather than write your own indentation function from scratch, it is often preferable to try and reuse some existing ones or to rely on a generic indentation engine. There are sadly few such engines. The CC-mode indentation code (used with C, C++, Java, Awk and a few other such modes) has been made more generic over the years, so if your language seems somewhat similar to one of those languages, you might try to use that engine. Another one is SMIE which takes an approach in the spirit of Lisp sexps and adapts it to non-Lisp languages. Yet another one is to rely on a full-blown parser, for example, the tree-sitter library.

24.7.1 Simple Minded Indentation Engine

SMIE is a package that provides a generic navigation and indentation engine. Based on a very simple parser using an operator precedence grammar, it lets major modes extend the sexp-based navigation of Lisp to non-Lisp languages as well as provide a simple to use but reliable auto-indentation.

Operator precedence grammar is a very primitive technology for parsing compared to some of the more common techniques used in compilers. It has the following characteristics: its parsing power is very limited, and it is largely unable to detect syntax errors, but it has the advantage of being algorithmically efficient and able to parse forward just as well as backward. In practice that means that SMIE can use it for indentation based on backward parsing, that it can provide both `forward-sexp` and `backward-sexp` functionality, and that it will naturally work on syntactically incorrect code without any extra effort. The downside is that it also means that most programming languages cannot be parsed correctly using SMIE, at least not without resorting to some special tricks (see Section 24.7.1.5 [SMIE Tricks], page 587).

24.7.1.1 SMIE Setup and Features

SMIE is meant to be a one-stop shop for structural navigation and various other features which rely on the syntactic structure of code, in particular automatic indentation. The main entry point is `smie-setup` which is a function typically called while setting up a major mode.

`smie-setup` *grammar rules-function* **&rest** *keywords* [Function]

Setup SMIE navigation and indentation. *grammar* is a grammar table generated by `smie-prec2->grammar`. *rules-function* is a set of indentation rules for use on `smie-rules-function`. *keywords* are additional arguments, which can include the following keywords:

- `:forward-token` *fun*: Specify the forward lexer to use.
- `:backward-token` *fun*: Specify the backward lexer to use.

Calling this function is sufficient to make commands such as `forward-sexp`, `backward-sexp`, and `transpose-sexps` be able to properly handle structural elements other than just the paired parentheses already handled by syntax tables. For example, if the provided grammar is precise enough, `transpose-sexps` can correctly transpose the two arguments of a `+` operator, taking into account the precedence rules of the language.

Calling `smie-setup` is also sufficient to make `TAB` indentation work in the expected way, extends `blink-matching-paren` to apply to elements like `begin...end`, and provides some commands that you can bind in the major mode keymap.

`smie-close-block` [Command]
This command closes the most recently opened (and not yet closed) block.

`smie-down-list` **&optional** *arg* [Command]
This command is like `down-list` but it also pays attention to nesting of tokens other than parentheses, such as `begin...end`.

24.7.1.2 Operator Precedence Grammars

SMIE's precedence grammars simply give to each token a pair of precedences: the left-precedence and the right-precedence. We say `T1 < T2` if the right-precedence of token `T1` is less than the left-precedence of token `T2`. A good way to read this `<` is as a kind of parenthesis: if we find `... T1 something T2 ...` then that should be parsed as `... T1 (something T2 ...` rather than as `... T1 something) T2 ...`. The latter interpretation would be the case if we had `T1 > T2`. If we have `T1 = T2`, it means that token T2 follows token T1 in the same syntactic construction, so typically we have `"begin" = "end"`. Such pairs of precedences are sufficient to express left-associativity or right-associativity of infix operators, nesting of tokens like parentheses and many other cases.

`smie-prec2->grammar` *table* [Function]
This function takes a *prec2* grammar *table* and returns an alist suitable for use in `smie-setup`. The *prec2 table* is itself meant to be built by one of the functions below.

`smie-merge-prec2s` **&rest** *tables* [Function]
This function takes several *prec2 tables* and merges them into a new *prec2* table.

`smie-precs->prec2` *precs* [Function]
This function builds a *prec2* table from a table of precedences *precs*. *precs* should be a list, sorted by precedence (for example `"+"` will come before `"*"`), of elements of the form `(assoc op ...)`, where each *op* is a token that acts as an operator; *assoc* is their associativity, which can be either `left`, `right`, `assoc`, or `nonassoc`. All operators in a given element share the same precedence level and associativity.

`smie-bnf->prec2` *bnf* **&rest** *resolvers* [Function]
This function lets you specify the grammar using a BNF notation. It accepts a *bnf* description of the grammar along with a set of conflict resolution rules *resolvers*, and returns a *prec2* table.

bnf is a list of nonterminal definitions of the form `(nonterm rhs1 rhs2 ...)` where each *rhs* is a (non-empty) list of terminals (aka tokens) or non-terminals.

Not all grammars are accepted:

- An *rhs* cannot be an empty list (an empty list is never needed, since SMIE allows all non-terminals to match the empty string anyway).
- An *rhs* cannot have 2 consecutive non-terminals: each pair of non-terminals needs to be separated by a terminal (aka token). This is a fundamental limitation of operator precedence grammars.

Additionally, conflicts can occur:

- The returned *prec2* table holds constraints between pairs of tokens, and for any given pair only one constraint can be present: T1 < T2, T1 = T2, or T1 > T2.
- A token can be an `opener` (something similar to an open-paren), a `closer` (like a close-paren), or `neither` of the two (e.g., an infix operator, or an inner token like `"else"`).

Precedence conflicts can be resolved via *resolvers*, which is a list of *precs* tables (see `smie-precs->prec2`): for each precedence conflict, if those `precs` tables specify a particular constraint, then the conflict is resolved by using this constraint instead, else a conflict is reported and one of the conflicting constraints is picked arbitrarily and the others are simply ignored.

24.7.1.3 Defining the Grammar of a Language

The usual way to define the SMIE grammar of a language is by defining a new global variable that holds the precedence table by giving a set of BNF rules. For example, the grammar definition for a small Pascal-like language could look like:

```
(require 'smie)
(defvar sample-smie-grammar
  (smie-prec2->grammar
   (smie-bnf->prec2
    '((id)
      (inst ("begin" insts "end")
            ("if" exp "then" inst "else" inst)
            (id ":=" exp)
            (exp))
      (insts (insts ";" insts) (inst))
      (exp (exp "+" exp)
           (exp "*" exp)
           ("(" exps ")"))
      (exps (exps "," exps) (exp)))
    '((assoc ";"))
    '((assoc ","))
    '((assoc "+") (assoc "*")))))
```

A few things to note:

- The above grammar does not explicitly mention the syntax of function calls: SMIE will automatically allow any sequence of sexps, such as identifiers, balanced parentheses, or `begin ... end` blocks to appear anywhere anyway.

- The grammar category `id` has no right hand side: this does not mean that it can match only the empty string, since as mentioned any sequence of sexps can appear anywhere anyway.
- Because non terminals cannot appear consecutively in the BNF grammar, it is difficult to correctly handle tokens that act as terminators, so the above grammar treats `";"` as a statement *separator* instead, which SMIE can handle very well.
- Separators used in sequences (such as `","` and `";"` above) are best defined with BNF rules such as `(foo (foo "separator" foo) ...)` which generate precedence conflicts which are then resolved by giving them an explicit `(assoc "separator")`.
- The `("(" exps ")")` rule was not needed to pair up parens, since SMIE will pair up any characters that are marked as having paren syntax in the syntax table. What this rule does instead (together with the definition of `exps`) is to make it clear that `","` should not appear outside of parentheses.
- Rather than have a single *precs* table to resolve conflicts, it is preferable to have several tables, so as to let the BNF part of the grammar specify relative precedences where possible.
- Unless there is a very good reason to prefer `left` or `right`, it is usually preferable to mark operators as associative, using `assoc`. For that reason `"+"` and `"*"` are defined above as `assoc`, although the language defines them formally as left associative.

24.7.1.4 Defining Tokens

SMIE comes with a predefined lexical analyzer which uses syntax tables in the following way: any sequence of characters that have word or symbol syntax is considered a token, and so is any sequence of characters that have punctuation syntax. This default lexer is often a good starting point but is rarely actually correct for any given language. For example, it will consider `"2,+3"` to be composed of 3 tokens: `"2"`, `",+"`, and `"3"`.

To describe the lexing rules of your language to SMIE, you need 2 functions, one to fetch the next token, and another to fetch the previous token. Those functions will usually first skip whitespace and comments and then look at the next chunk of text to see if it is a special token. If so it should skip the token and return a description of this token. Usually this is simply the string extracted from the buffer, but it can be anything you want. For example:

```
(defvar sample-keywords-regexp
  (regexp-opt '("+" "*" "," ";" ">" ">=" "<" "<=" ":=" "=")))
(defun sample-smie-forward-token ()
  (forward-comment (point-max))
  (cond
   ((looking-at sample-keywords-regexp)
    (goto-char (match-end 0))
    (match-string-no-properties 0))
   (t (buffer-substring-no-properties
       (point)
       (progn (skip-syntax-forward "w_")
              (point))))))
```

```
(defun sample-smie-backward-token ()
  (forward-comment (- (point)))
  (cond
   ((looking-back sample-keywords-regexp (- (point) 2) t)
    (goto-char (match-beginning 0))
    (match-string-no-properties 0))
   (t (buffer-substring-no-properties
       (point)
       (progn (skip-syntax-backward "w_")
              (point))))))
```

Notice how those lexers return the empty string when in front of parentheses. This is because SMIE automatically takes care of the parentheses defined in the syntax table. More specifically if the lexer returns `nil` or an empty string, SMIE tries to handle the corresponding text as a sexp according to syntax tables.

24.7.1.5 Living With a Weak Parser

The parsing technique used by SMIE does not allow tokens to behave differently in different contexts. For most programming languages, this manifests itself by precedence conflicts when converting the BNF grammar.

Sometimes, those conflicts can be worked around by expressing the grammar slightly differently. For example, for Modula-2 it might seem natural to have a BNF grammar that looks like this:

```
...
(inst ("IF" exp "THEN" insts "ELSE" insts "END")
      ("CASE" exp "OF" cases "END")
      ...)
(cases (cases "|" cases)
       (caselabel ":" insts)
       ("ELSE" insts))
...
```

But this will create conflicts for `"ELSE"`: on the one hand, the IF rule implies (among many other things) that `"ELSE" = "END"`; but on the other hand, since `"ELSE"` appears within `cases`, which appears left of `"END"`, we also have `"ELSE" > "END"`. We can solve the conflict either by using:

```
...
(inst ("IF" exp "THEN" insts "ELSE" insts "END")
      ("CASE" exp "OF" cases "END")
      ("CASE" exp "OF" cases "ELSE" insts "END")
      ...)
(cases (cases "|" cases) (caselabel ":" insts))
...
```

or

```
...
(inst ("IF" exp "THEN" else "END")
      ("CASE" exp "OF" cases "END")
```

```
      ...)
(else (insts "ELSE" insts))
(cases (cases "|" cases) (caselabel ":" insts) (else))
...
```

Reworking the grammar to try and solve conflicts has its downsides, tho, because SMIE assumes that the grammar reflects the logical structure of the code, so it is preferable to keep the BNF closer to the intended abstract syntax tree.

Other times, after careful consideration you may conclude that those conflicts are not serious and simply resolve them via the *resolvers* argument of `smie-bnf->prec2`. Usually this is because the grammar is simply ambiguous: the conflict does not affect the set of programs described by the grammar, but only the way those programs are parsed. This is typically the case for separators and associative infix operators, where you want to add a resolver like `'((assoc "|"))`. Another case where this can happen is for the classic *dangling else* problem, where you will use `'((assoc "else" "then"))`. It can also happen for cases where the conflict is real and cannot really be resolved, but it is unlikely to pose a problem in practice.

Finally, in many cases some conflicts will remain despite all efforts to restructure the grammar. Do not despair: while the parser cannot be made more clever, you can make the lexer as smart as you want. So, the solution is then to look at the tokens involved in the conflict and to split one of those tokens into 2 (or more) different tokens. E.g., if the grammar needs to distinguish between two incompatible uses of the token `"begin"`, make the lexer return different tokens (say `"begin-fun"` and `"begin-plain"`) depending on which kind of `"begin"` it finds. This pushes the work of distinguishing the different cases to the lexer, which will thus have to look at the surrounding text to find ad-hoc clues.

24.7.1.6 Specifying Indentation Rules

Based on the provided grammar, SMIE will be able to provide automatic indentation without any extra effort. But in practice, this default indentation style will probably not be good enough. You will want to tweak it in many different cases.

SMIE indentation is based on the idea that indentation rules should be as local as possible. To this end, it relies on the idea of *virtual* indentation, which is the indentation that a particular program point would have if it were at the beginning of a line. Of course, if that program point is indeed at the beginning of a line, its virtual indentation is its current indentation. But if not, then SMIE uses the indentation algorithm to compute the virtual indentation of that point. Now in practice, the virtual indentation of a program point does not have to be identical to the indentation it would have if we inserted a newline before it. To see how this works, the SMIE rule for indentation after a `{` in C does not care whether the `{` is standing on a line of its own or is at the end of the preceding line. Instead, these different cases are handled in the indentation rule that decides how to indent before a `{`.

Another important concept is the notion of *parent*: The *parent* of a token, is the head token of the nearest enclosing syntactic construct. For example, the parent of an `else` is the `if` to which it belongs, and the parent of an `if`, in turn, is the lead token of the surrounding construct. The command `backward-sexp` jumps from a token to its parent, but there are some caveats: for *openers* (tokens which start a construct, like `if`), you need to start with point before the token, while for others you need to start with point after the

token. `backward-sexp` stops with point before the parent token if that is the *opener* of the token of interest, and otherwise it stops with point after the parent token.

SMIE indentation rules are specified using a function that takes two arguments *method* and *arg* where the meaning of *arg* and the expected return value depend on *method*.

method can be:

- `:after`, in which case *arg* is a token and the function should return the *offset* to use for indentation after *arg*.
- `:before`, in which case *arg* is a token and the function should return the *offset* to use to indent *arg* itself.
- `:elem`, in which case the function should return either the offset to use to indent function arguments (if *arg* is the symbol `arg`) or the basic indentation step (if *arg* is the symbol `basic`).
- `:list-intro`, in which case *arg* is a token and the function should return non-`nil` if the token is followed by a list of expressions (not separated by any token) rather than an expression.

When *arg* is a token, the function is called with point just before that token. A return value of `nil` always means to fallback on the default behavior, so the function should return `nil` for arguments it does not expect.

offset can be:

- `nil`: use the default indentation rule.
- `(column . column)`: indent to column *column*.
- *number*: offset by *number*, relative to a base token which is the current token for `:after` and its parent for `:before`.

24.7.1.7 Helper Functions for Indentation Rules

SMIE provides various functions designed specifically for use in the indentation rules function (several of those functions break if used in another context). These functions all start with the prefix `smie-rule-`.

`smie-rule-bolp` [Function]
Return non-`nil` if the current token is the first on the line.

`smie-rule-hanging-p` [Function]
Return non-`nil` if the current token is *hanging*. A token is *hanging* if it is the last token on the line and if it is preceded by other tokens: a lone token on a line is not hanging.

`smie-rule-next-p` **&rest** *tokens* [Function]
Return non-`nil` if the next token is among *tokens*.

`smie-rule-prev-p` **&rest** *tokens* [Function]
Return non-`nil` if the previous token is among *tokens*.

`smie-rule-parent-p` **&rest** *parents* [Function]
Return non-`nil` if the current token's parent is among *parents*.

`smie-rule-sibling-p` [Function]
Return non-`nil` if the current token's parent is actually a sibling. This is the case for example when the parent of a `","` is just the previous `","`.

`smie-rule-parent` **&optional** *offset* [Function]
Return the proper offset to align the current token with the parent. If non-`nil`, *offset* should be an integer giving an additional offset to apply.

`smie-rule-separator` *method* [Function]
Indent current token as a *separator*.

By *separator*, we mean here a token whose sole purpose is to separate various elements within some enclosing syntactic construct, and which does not have any semantic significance in itself (i.e., it would typically not exist as a node in an abstract syntax tree).

Such a token is expected to have an associative syntax and be closely tied to its syntactic parent. Typical examples are `","` in lists of arguments (enclosed inside parentheses), or `";"` in sequences of instructions (enclosed in a `{...}` or `begin...end` block).

method should be the method name that was passed to `smie-rules-function`.

24.7.1.8 Sample Indentation Rules

Here is an example of an indentation function:

```
(defun sample-smie-rules (kind token)
  (pcase (cons kind token)
    (`(:elem . basic) sample-indent-basic)
    (`(,_ . ",") (smie-rule-separator kind))
    (`(:after . ":=") sample-indent-basic)
    (`(:before . ,(or `"begin" `"(" `"{"))
     (if (smie-rule-hanging-p) (smie-rule-parent)))
    (`(:before . "if")
     (and (not (smie-rule-bolp)) (smie-rule-prev-p "else")
          (smie-rule-parent)))))
```

A few things to note:

- The first case indicates the basic indentation increment to use. If `sample-indent-basic` is `nil`, then SMIE uses the global setting `smie-indent-basic`. The major mode could have set `smie-indent-basic` buffer-locally instead, but that is discouraged.
- The rule for the token `","` make SMIE try to be more clever when the comma separator is placed at the beginning of lines. It tries to outdent the separator so as to align the code after the comma; for example:

```
x = longfunctionname (
        arg1
      , arg2
    );
```

- The rule for indentation after `":="` exists because otherwise SMIE would treat `":="` as an infix operator and would align the right argument with the left one.

- The rule for indentation before `"begin"` is an example of the use of virtual indentation: This rule is used only when `"begin"` is hanging, which can happen only when `"begin"` is not at the beginning of a line. So this is not used when indenting `"begin"` itself but only when indenting something relative to this `"begin"`. Concretely, this rule changes the indentation from:

```
if x > 0 then begin
        dosomething(x);
    end
```

  to

```
if x > 0 then begin
    dosomething(x);
end
```

- The rule for indentation before `"if"` is similar to the one for `"begin"`, but where the purpose is to treat `"else if"` as a single unit, so as to align a sequence of tests rather than indent each test further to the right. This function does this only in the case where the `"if"` is not placed on a separate line, hence the `smie-rule-bolp` test.

  If we know that the `"else"` is always aligned with its `"if"` and is always at the beginning of a line, we can use a more efficient rule:

```
((equal token "if")
 (and (not (smie-rule-bolp))
      (smie-rule-prev-p "else")
      (save-excursion
        (sample-smie-backward-token)
        (cons 'column (current-column)))))
```

  The advantage of this formulation is that it reuses the indentation of the previous `"else"`, rather than going all the way back to the first `"if"` of the sequence.

24.7.1.9 Customizing Indentation

If you are using a mode whose indentation is provided by SMIE, you can customize the indentation to suit your preferences. You can do this on a per-mode basis (using the option `smie-config`), or a per-file basis (using the function `smie-config-local` in a file-local variable specification).

`smie-config` [User Option]

This option lets you customize indentation on a per-mode basis. It is an alist with elements of the form `(`*`mode . rules`*`)`. For the precise form of rules, see the variable's documentation; but you may find it easier to use the command `smie-config-guess`.

`smie-config-guess` [Command]

This command tries to work out appropriate settings to produce your preferred style of indentation. Simply call the command while visiting a file that is indented with your style.

`smie-config-save` [Command]

Call this command after using `smie-config-guess`, to save your settings for future sessions.

`smie-config-show-indent` **&optional** *move* [Command]
This command displays the rules that are used to indent the current line.

`smie-config-set-indent` [Command]
This command adds a local rule to adjust the indentation of the current line.

`smie-config-local` *rules* [Function]
This function adds *rules* as indentation rules for the current buffer. These add to any mode-specific rules defined by the `smie-config` option. To specify custom indentation rules for a specific file, add an entry to the file's local variables of the form: `eval: (smie-config-local '(rules))`.

24.7.2 Parser-based Indentation

When built with the tree-sitter library (see Chapter 37 [Parsing Program Source], page 1042), Emacs is capable of parsing the program source and producing a syntax tree. This syntax tree can be used for guiding the program source indentation commands. For maximum flexibility, it is possible to write a custom indentation function that queries the syntax tree and indents accordingly for each language, but that is a lot of work. It is more convenient to use the simple indentation engine described below: then the major mode needs only write some indentation rules, and the engine takes care of the rest.

To enable the parser-based indentation engine, either set `treesit-simple-indent-rules` and call `treesit-major-mode-setup`, or equivalently, set the value of `indent-line-function` to `treesit-indent`.

`treesit-indent-function` [Variable]
This variable stores the actual function called by `treesit-indent`. By default, its value is `treesit-simple-indent`. In the future we might add other, more complex indentation engines.

Writing indentation rules

`treesit-simple-indent-rules` [Variable]
This local variable stores indentation rules for every language. It is an alist with elements of the form `(language . rules)`, where *language* is a language symbol, and *rules* is a list with elements of the form `(matcher anchor offset)`.

First, Emacs passes the smallest tree-sitter node at the beginning of the current line to *matcher*; if it returns non-`nil`, this rule is applicable. Then Emacs passes the node to *anchor*, which returns a buffer position. Emacs takes the column number of that position, adds *offset* to it, and the result is the indentation column for the current line.

The *matcher* and *anchor* are functions, and Emacs provides convenient defaults for them.

Each *matcher* or *anchor* is a function that takes three arguments: *node*, *parent*, and *bol*. The argument *bol* is the buffer position whose indentation is required: the position of the first non-whitespace character after the beginning of the line. The argument *node* is the largest node that starts at that position (and is not a root node); and *parent* is the parent of *node*. However, when that position is in a whitespace or

inside a multi-line string, no node can start at that position, so *node* is `nil`. In that case, *parent* would be the smallest node that spans that position.

matcher should return non-`nil` if the rule is applicable, and *anchor* should return a buffer position.

offset can be an integer, a variable whose value is an integer, or a function that returns an integer. If it is a function, it is passed *node*, *parent*, and *bol*, like matchers and anchors.

`treesit-simple-indent-presets` [Variable]

This is a list of defaults for *matcher*s and *anchor*s in `treesit-simple-indent-rules`. Each of them represents a function that takes 3 arguments: *node*, *parent*, and *bol*. The available default functions are:

`no-node` This matcher is a function that is called with 3 arguments: *node*, *parent*, and *bol*. It returns non-`nil`, indicating a match, if *node* is `nil`, i.e., there is no node that starts at *bol*. This is the case when *bol* is on an empty line or inside a multi-line string, etc.

`parent-is`
This matcher is a function of one argument, *type*; it returns a function that is called with 3 arguments: *node*, *parent*, and *bol*, and returns non-`nil` (i.e., a match) if *parent*'s type matches regexp *type*.

`node-is` This matcher is a function of one argument, *type*; it returns a function that is called with 3 arguments: *node*, *parent*, and *bol*, and returns non-`nil` if *node*'s type matches regexp *type*.

`field-is` This matcher is a function of one argument, *name*; it returns a function that is called with 3 arguments: *node*, *parent*, and *bol*, and returns non-`nil` if *node*'s field name in *parent* matches regexp *name*.

`query` This matcher is a function of one argument, *query*; it returns a function that is called with 3 arguments: *node*, *parent*, and *bol*, and returns non-`nil` if querying *parent* with *query* captures *node* (see Section 37.5 [Pattern Matching], page 1055).

`match` This matcher is a function of 5 arguments: *node-type*, *parent-type*, *node-field*, *node-index-min*, and *node-index-max*). It returns a function that is called with 3 arguments: *node*, *parent*, and *bol*, and returns non-`nil` if *node*'s type matches regexp *node-type*, *parent*'s type matches regexp *parent-type*, *node*'s field name in *parent* matches regexp *node-field*, and *node*'s index among its siblings is between *node-index-min* and *node-index-max*. If the value of an argument is `nil`, this matcher doesn't check that argument. For example, to match the first child where parent is `argument_list`, use

```
(match nil "argument_list" nil 0 0)
```

In addition, *node-type* can be a special value `null`, which matches when the value of *node* is `nil`.

`n-p-gp` Short for "node-parent-grandparent", this matcher is a function of 3 arguments: *node-type*, *parent-type*, and *grandparent-type*. It returns a

function that is called with 3 arguments: *node*, *parent*, and *bol*, and returns non-`nil` if: (1) *node-type* matches *node*'s type, and (2) *parent-type* matches *parent*'s type, and (3) *grandparent-type* matches *parent*'s parent's type. If any of *node-type*, *parent-type*, and *grandparent-type* is `nil`, this function doesn't check for it.

`comment-end`
: This matcher is a function that is called with 3 arguments: *node*, *parent*, and *bol*, and returns non-`nil` if point is before a comment-ending token. Comment-ending tokens are defined by regexp `comment-end-skip`.

`catch-all`
: This matcher is a function that is called with 3 arguments: *node*, *parent*, and *bol*. It always returns non-`nil`, indicating a match.

`first-sibling`
: This anchor is a function that is called with 3 arguments: *node*, *parent*, and *bol*, and returns the start of the first child of *parent*.

`nth-sibling`
: This anchor is a function of two arguments: *n*, and an optional argument *named*. It returns a function that is called with 3 arguments: *node*, *parent*, and *bol*, and returns the start of the *n*th child of *parent*. If *named* is non-`nil`, only named children are counted (see [tree-sitter named node], page 1044).

`parent`
: This anchor is a function that is called with 3 arguments: *node*, *parent*, and *bol*, and returns the start of *parent*.

`grand-parent`
: This anchor is a function that is called with 3 arguments: *node*, *parent*, and *bol*, and returns the start of *parent*'s parent.

`great-grand-parent`
: This anchor is a function that is called with 3 arguments: *node*, *parent*, and *bol*, and returns the start of *parent*'s parent's parent.

`parent-bol`
: This anchor is a function that is called with 3 arguments: *node*, *parent*, and *bol*, and returns the first non-space character on the line which *parent*'s start is on.

`standalone-parent`
: This anchor is a function that is called with 3 arguments: *node*, *parent*, and *bol*. It finds the first ancestor node (parent, grandparent, etc.) of *node* that starts on its own line, and return the start of that node. "Starting on its own line" means there is only whitespace character before the node on the line which the node's start is on.

`prev-sibling`
: This anchor is a function that is called with 3 arguments: *node*, *parent*, and *bol*, and returns the start of the previous sibling of *node*.

`no-indent`
: This anchor is a function that is called with 3 arguments: *node*, *parent*, and *bol*, and returns the start of *node*.

`prev-line`
: This anchor is a function that is called with 3 arguments: *node*, *parent*, and *bol*, and returns the first non-whitespace character on the previous line.

`column-0`
: This anchor is a function that is called with 3 arguments: *node*, *parent*, and *bol*, and returns the beginning of the current line, which is at column 0.

`comment-start`
: This anchor is a function that is called with 3 arguments: *node*, *parent*, and *bol*, and returns the position after the comment-start token. Comment-start tokens are defined by regular expression `comment-start-skip`. This function assumes *parent* is the comment node.

`prev-adaptive-prefix`
: This anchor is a function that is called with 3 arguments: *node*, *parent*, and *bol*. It tries to match `adaptive-fill-regexp` to the text at the beginning of the previous non-empty line. If there is a match, this function returns the end of the match, otherwise it returns `nil`. However, if the current line begins with a prefix (e.g., '`-`'), return the beginning of the prefix of the previous line instead, so that the two prefixes align. This anchor is useful for an `indent-relative`-like indent behavior for block comments.

Indentation utilities

Here are some utility functions that can help writing parser-based indentation rules.

`treesit-check-indent` *mode* [Command]
: This command checks the current buffer's indentation against major mode *mode*. It indents the current buffer according to *mode* and compares the results with the current indentation. Then it pops up a buffer showing the differences. Correct indentation (target) is shown in green color, current indentation is shown in red color.

It is also helpful to use `treesit-inspect-mode` (see Section 37.1 [Language Grammar], page 1042) when writing indentation rules.

24.8 Desktop Save Mode

Desktop Save Mode is a feature to save the state of Emacs from one session to another. The user-level commands for using Desktop Save Mode are described in the GNU Emacs Manual (see Section "Saving Emacs Sessions" in *the GNU Emacs Manual*). Modes whose buffers visit a file, don't have to do anything to use this feature.

For buffers not visiting a file to have their state saved, the major mode must bind the buffer local variable `desktop-save-buffer` to a non-`nil` value.

`desktop-save-buffer` [Variable]

If this buffer-local variable is non-`nil`, the buffer will have its state saved in the desktop file at desktop save. If the value is a function, it is called at desktop save with argument *desktop-dirname*, and its value is saved in the desktop file along with the state of the buffer for which it was called. When file names are returned as part of the auxiliary information, they should be formatted using the call

```
(desktop-file-name file-name desktop-dirname)
```

For buffers not visiting a file to be restored, the major mode must define a function to do the job, and that function must be listed in the alist `desktop-buffer-mode-handlers`.

`desktop-buffer-mode-handlers` [Variable]

Alist with elements

```
(major-mode . restore-buffer-function)
```

The function *restore-buffer-function* will be called with argument list

```
(buffer-file-name buffer-name desktop-buffer-misc)
```

and it should return the restored buffer. Here *desktop-buffer-misc* is the value returned by the function optionally bound to `desktop-save-buffer`.

25 Documentation

GNU Emacs has convenient built-in help facilities, most of which derive their information from documentation strings associated with functions and variables. This chapter describes how to access documentation strings in Lisp programs.

The contents of a documentation string should follow certain conventions. In particular, its first line should be a complete sentence (or two complete sentences) that briefly describes what the function or variable does. See Section D.6 [Documentation Tips], page 1342, for how to write good documentation strings.

Note that the documentation strings for Emacs are not the same thing as the Emacs manual. Manuals have their own source files, written in the Texinfo language; documentation strings are specified in the definitions of the functions and variables they apply to. A collection of documentation strings is not sufficient as a manual because a good manual is not organized in that fashion; it is organized in terms of topics of discussion.

For commands to display documentation strings, see Section "Help" in *The GNU Emacs Manual*.

25.1 Documentation Basics

A documentation string is written using the Lisp syntax for strings, with double-quote characters surrounding the text. It is, in fact, an actual Lisp string. When the string appears in the proper place in a function or variable definition, it serves as the function's or variable's documentation.

In a function definition (a `lambda` or `defun` form), the documentation string is specified after the argument list, and is normally stored directly in the function object. See Section 13.2.4 [Function Documentation], page 235. You can also put function documentation in the `function-documentation` property of a function name (see Section 25.2 [Accessing Documentation], page 598).

In a variable definition (a `defvar` form), the documentation string is specified after the initial value. See Section 12.5 [Defining Variables], page 194. The string is stored in the variable's `variable-documentation` property.

Sometimes, Emacs does not keep documentation strings in memory. There are two such circumstances. Firstly, to save memory, the documentation for primitive functions (see Section 13.1 [What Is a Function], page 230) and built-in variables is kept in a file named `DOC`, in the directory specified by `doc-directory` (see Section 25.2 [Accessing Documentation], page 598). Secondly, when a function or variable is loaded from a byte-compiled file, Emacs avoids loading its documentation string (see Section 17.3 [Docs and Compilation], page 320). In both cases, Emacs looks up the documentation string from the file only when needed, such as when the user calls *`C-h f`* (`describe-function`) for a function.

Documentation strings can contain special *key substitution sequences*, referring to key bindings which are looked up only when the user views the documentation. This allows the help commands to display the correct keys even if a user rearranges the default key bindings. See Section 25.3 [Keys in Documentation], page 600.

In the documentation string of an autoloaded command (see Section 16.5 [Autoload], page 305), these key-substitution sequences have an additional special effect: they cause

`C-h f` on the command to trigger autoloading. (This is needed for correctly setting up the hyperlinks in the `*Help*` buffer.)

25.2 Access to Documentation Strings

`documentation-property` *symbol property* **&optional** *verbatim* [Function]

This function returns the documentation string recorded in *symbol*'s property list under property *property*. It is most often used to look up the documentation strings of variables, for which *property* is `variable-documentation`. However, it can also be used to look up other kinds of documentation, such as for customization groups (but for function documentation, use the `documentation` function, below).

If the property value refers to a documentation string stored in the `DOC` file or a byte-compiled file, this function looks up that string and returns it.

If the property value isn't `nil`, isn't a string, and doesn't refer to text in a file, then it is evaluated as a Lisp expression to obtain a string.

Finally, this function passes the string through `substitute-command-keys` to substitute key bindings (see Section 25.3 [Keys in Documentation], page 600). It skips this step if *verbatim* is non-`nil`.

```
(documentation-property 'command-line-processed
   'variable-documentation)
     ⇒ "Non-nil once command line has been processed"
(symbol-plist 'command-line-processed)
     ⇒ (variable-documentation 188902)
(documentation-property 'emacs 'group-documentation)
     ⇒ "Customization of the One True Editor."
```

`documentation` *function* **&optional** *verbatim* [Function]

This function returns the documentation string of *function*. It handles macros, named keyboard macros, and special forms, as well as ordinary functions.

If *function* is a symbol, this function first looks for the `function-documentation` property of that symbol; if that has a non-`nil` value, the documentation comes from that value (if the value is not a string, it is evaluated).

If *function* is not a symbol, or if it has no `function-documentation` property, then `documentation` extracts the documentation string from the actual function definition, reading it from a file if called for.

Finally, unless *verbatim* is non-`nil`, this function calls `substitute-command-keys`. The result is the documentation string to return.

The `documentation` function signals a `void-function` error if *function* has no function definition. However, it is OK if the function definition has no documentation string. In that case, `documentation` returns `nil`.

`function-documentation` *function* [Function]

Generic function used by `documentation` to extract the raw docstring from a function object. You can specify how to get the docstring of a specific function type by adding a corresponding method to it.

`face-documentation` *face* [Function]

This function returns the documentation string of *face* as a face.

Here is an example of using the two functions, `documentation` and `documentation-property`, to display the documentation strings for several symbols in a `*Help*` buffer.

```
(defun describe-symbols (pattern)
  "Describe the Emacs Lisp symbols matching PATTERN.
All symbols that have PATTERN in their name are described
in the *Help* buffer."
  (interactive "sDescribe symbols matching: ")
  (let ((describe-func
         (lambda (s)
           ;; Print description of symbol.
           (if (fboundp s)             ; It is a function.
               (princ
                (format "%s\t%s\n%s\n\n" s
                  (if (commandp s)
                      (let ((keys (where-is-internal s)))
                        (if keys
                            (concat
                             "Keys: "
                             (mapconcat 'key-description
                                        keys " "))
                          "Keys: none"))
                    "Function")
                  (or (documentation s)
                      "not documented"))))

           (if (boundp s)              ; It is a variable.
               (princ
                (format "%s\t%s\n%s\n\n" s
                  (if (custom-variable-p s)
                      "Option " "Variable")
                  (or (documentation-property
                       s 'variable-documentation)
                      "not documented"))))))
        sym-list)

    ;; Build a list of symbols that match pattern.
    (mapatoms (lambda (sym)
                (if (string-match pattern (symbol-name sym))
                    (setq sym-list (cons sym sym-list)))))

    ;; Display the data.
    (help-setup-xref (list 'describe-symbols pattern)
                     (called-interactively-p 'interactive))
    (with-help-window (help-buffer)
      (mapcar describe-func (sort sym-list 'string<)))))
```

The `describe-symbols` function works like `apropos`, but provides more information.

```
(describe-symbols "goal")

---------- Buffer: *Help* ----------
goal-column     Option
Semipermanent goal column for vertical motion, as set by ...

minibuffer-temporary-goal-position      Variable
not documented
```

```
set-goal-column Keys: C-x C-n
Set the current horizontal position as a goal for C-n and C-p.
Those commands will move to this position in the line moved to
rather than trying to keep the same horizontal position.
With a non-nil argument ARG, clears out the goal column
so that C-n and C-p resume vertical motion.
The goal column is stored in the variable ‘goal-column’.

(fn ARG)

temporary-goal-column   Variable
Current goal column for vertical motion.
It is the column where point was at the start of the current run
of vertical motion commands.

When moving by visual lines via the function ‘line-move-visual’, it is a cons
cell (COL . HSCROLL), where COL is the x-position, in pixels,
divided by the default column width, and HSCROLL is the number of
columns by which window is scrolled from left margin.

When the ‘track-eol’ feature is doing its job, the value is
‘most-positive-fixnum’.
---------- Buffer: *Help* ----------
```

`Snarf-documentation` *filename* [Function]

This function is used when building Emacs, just before the runnable Emacs is dumped. It finds the positions of the documentation strings stored in the file *filename*, and records those positions into memory in the function definitions and variable property lists. See Section E.1 [Building Emacs], page 1351.

Emacs reads the file *filename* from the `emacs/etc` directory. When the dumped Emacs is later executed, the same file will be looked for in the directory `doc-directory`. Usually *filename* is `"DOC"`.

`doc-directory` [Variable]

This variable holds the name of the directory which should contain the file `"DOC"` that contains documentation strings for built-in functions and variables.

In most cases, this is the same as `data-directory`. They may be different when you run Emacs from the directory where you built it, without actually installing it. See [Definition of data-directory], page 606.

25.3 Substituting Key Bindings in Documentation

When documentation strings refer to key sequences, they should use the current, actual key bindings. They can do so using certain special text sequences described below. Accessing documentation strings in the usual way substitutes current key binding information for these special sequences. This works by calling `substitute-command-keys`. You can also call that function yourself.

Here is a list of the special sequences and what they mean:

`\[`*command*`]`

stands for a key sequence that will invoke *command*, or ‘`M-x` *command*’ if *command* has no key bindings.

`\{`*mapvar*`}`
: stands for a summary of the keymap which is the value of the variable *mapvar*. The summary is made using `describe-bindings`. The summary will normally exclude meny bindings, but if the *include-menus* argument to `substitute-command-keys` is non-`nil`, the menu bindings will be included.

`\<`*mapvar*`>`
: stands for no text itself. It is used only for a side effect: it specifies *mapvar*'s value as the keymap for any following '`\[`*command*`]`' sequences in this documentation string.

`\``*KEYSEQ*`'`
: stands for a key sequence *KEYSEQ*, which will use the same face as a command substitution. This should be used only when a key sequence has no corresponding command, for example when it is read directly with `read-key-sequence`. It must be a valid key sequence according to `key-valid-p`. It can also be used with command names, like '`\`M-x foo'`', where you want this to be fontified like a keyboard sequence, but you want to inhibit translating it into a key sequence like '`\[foo]`' does.

`` ` ``
: (grave accent) stands for a left quote. This generates a left single quotation mark, an apostrophe, or a grave accent depending on the value of `text-quoting-style`. See Section 25.4 [Text Quoting Style], page 602.

`'`
: (apostrophe) stands for a right quote. This generates a right single quotation mark or an apostrophe depending on the value of `text-quoting-style`.

`\=`
: quotes the following character and is discarded; thus, '`\=``' puts '`` ' into the output, '`\=\[`' puts '`\[`' into the output, and '`\=\=`' puts '`\=`' into the output.

`\+`
: This indicates that the symbol directly following should not be marked as link in the `*Help*` buffer.

Please note: Each '`\`' must be doubled when written in a string in Emacs Lisp.

`substitute-command-keys` *string* **&optional** *no-face include-menus* [Function]
: This function scans *string* for the above special sequences and replaces them by what they stand for, returning the result as a string. This permits display of documentation that refers accurately to the user's own customized key bindings. By default, the key bindings are given a special face `help-key-binding`, but if the optional argument *no-face* is non-`nil`, the function doesn't add this face to the produced string.

  If a command has multiple bindings, this function normally uses the first one it finds. You can specify one particular key binding by assigning an `:advertised-binding` symbol property to the command, like this:

```
(put 'undo :advertised-binding [?\C-/])
```

  The `:advertised-binding` property also affects the binding shown in menu items (see Section 23.18.5 [Menu Bar], page 517). The property is ignored if it specifies a key binding that the command does not actually have.

Here are examples of the special sequences:

```
(substitute-command-keys
   "To abort recursive edit, type `\\[abort-recursive-edit]'.")
⇒ "To abort recursive edit, type ‘C-]’."
```

```
(substitute-command-keys
   "The keys that are defined for the minibuffer here are:
  \\{minibuffer-local-must-match-map}")
⇒ "The keys that are defined for the minibuffer here are:

?               minibuffer-completion-help
SPC             minibuffer-complete-word
TAB             minibuffer-complete
C-j             minibuffer-complete-and-exit
RET             minibuffer-complete-and-exit
C-g             abort-recursive-edit
"

The keymap description will normally exclude menu items, but if
include-menus is non-nil, include them.

(substitute-command-keys
   "To abort a recursive edit from the minibuffer, type \
`\\<minibuffer-local-must-match-map>\\[abort-recursive-edit]'.")
⇒ "To abort a recursive edit from the minibuffer, type ‘C-g’."
```

`substitute-quotes` *string* [Function]

This function works like `substitute-command-keys`, but only replaces quote characters.

There are other special conventions for the text in documentation strings—for instance, you can refer to functions, variables, and sections of this manual. See Section D.6 [Documentation Tips], page 1342, for details.

25.4 Text Quoting Style

Typically, grave accents and apostrophes are treated specially in documentation strings and diagnostic messages, and translate to matching single quotation marks (also called "curved quotes"). For example, the documentation string `"Alias for ‘foo’."` and the function call `(message "Alias for `foo'.")` both translate to `"Alias for ‘foo’."`. Less commonly, Emacs displays grave accents and apostrophes as themselves, or as apostrophes only (e.g., `"Alias for 'foo'."`). Documentation strings and message formats should be written so that they display well with any of these styles. For example, the documentation string `"Alias for 'foo'."` is probably not what you want, as it can display as `"Alias for 'foo'."`, an unusual style in English.

Sometimes you may need to display a grave accent or apostrophe without translation, regardless of text quoting style. In a documentation string, you can do this with escapes. For example, in the documentation string `"\\=‘(a ,(sin 0)) ==> (a 0.0)"` the grave accent is intended to denote Lisp code, so it is escaped and displays as itself regardless of quoting style. In a call to `message` or `error`, you can avoid translation by using a format `"%s"` with an argument that is a call to `format`. For example, `(message "%s" (format "`(a ,(sin %S)) ==> (a %S)" x (sin x)))` displays a message that starts with grave accent regardless of text quoting style.

`text-quoting-style` [User Option]

The value of this user option is a symbol that specifies the style Emacs should use for single quotes in the wording of help and messages. If the option's value is `curve`,

the style is ‘`like this`’ with curved single quotes. If the value is `straight`, the style is ’`like this`’ with straight apostrophes. If the value is `grave`, quotes are not translated and the style is ‘`like this`’ with grave accent and apostrophe, the standard style before Emacs version 25. The default value `nil` acts like `curve` if curved single quotes seem to be displayable, and like `grave` otherwise.

This option is useful on platforms that have problems with curved quotes. You can customize it freely according to your personal preference.

`text-quoting-style` [Function]
You should not read the value of the variable `text-quoting-style` directly. Instead, use this function with the same name to dynamically compute the correct quoting style on the current terminal in the `nil` case described above.

25.5 Describing Characters for Help Messages

These functions convert events, key sequences, or characters to textual descriptions. These descriptions are useful for including arbitrary text characters or key sequences in messages, because they convert non-printing and whitespace characters to sequences of printing characters. The description of a non-whitespace printing character is the character itself.

`key-description` *sequence* **&optional** *prefix* [Function]
This function returns a string containing the Emacs standard notation for the input events in *sequence*. If *prefix* is non-`nil`, it is a sequence of input events leading up to *sequence* and is included in the return value. Both arguments may be strings, vectors or lists. See Section 22.7 [Input Events], page 441, for more information about valid events.

```
(key-description [?\M-3 delete])
     ⇒ "M-3 <delete>"
(key-description [delete] "\M-3")
     ⇒ "M-3 <delete>"
```

See also the examples for `single-key-description`, below.

`single-key-description` *event* **&optional** *no-angles* [Function]
This function returns a string describing *event* in the standard Emacs notation for keyboard input. A normal printing character appears as itself, but a control character turns into a string starting with ‘`C-`’, a meta character turns into a string starting with ‘`M-`’, and space, tab, etc., appear as ‘`SPC`’, ‘`TAB`’, etc. A function key symbol appears inside angle brackets ‘`<...>`’. An event that is a list appears as the name of the symbol in the CAR of the list, inside angle brackets.

If the optional argument *no-angles* is non-`nil`, the angle brackets around function keys and event symbols are omitted; this is for compatibility with old versions of Emacs which didn't use the brackets.

```
(single-key-description ?\C-x)
     ⇒ "C-x"
(key-description "\C-x \M-y \n \t \r \f123")
     ⇒ "C-x SPC M-y SPC C-j SPC TAB SPC RET SPC C-l 1 2 3"
(single-key-description 'delete)
     ⇒ "<delete>"
(single-key-description 'C-mouse-1)
     ⇒ "C-<mouse-1>"
```

```
(single-key-description 'C-mouse-1 t)
     ⇒ "C-mouse-1"
```

`text-char-description` *character* [Function]

This function returns a string describing *character* in the standard Emacs notation for characters that can appear in text—similar to `single-key-description`, except that the argument must be a valid character code that passes a `characterp` test (see Section 34.5 [Character Codes], page 970). The function produces descriptions of control characters with a leading caret (which is how Emacs usually displays control characters in buffers). Characters with modifier bits will cause this function to signal an error (ASCII characters with the Control modifier are an exception, they are represented as control characters).

```
(text-char-description ?\C-c)
     ⇒ "^C"
(text-char-description ?\M-m)
     error  Wrong type argument: characterp, 134217837
```

`read-kbd-macro` *string* **&optional** *need-vector* [Command]

This function is used mainly for operating on keyboard macros, but it can also be used as a rough inverse for `key-description`. You call it with a string containing key descriptions, separated by spaces; it returns a string or vector containing the corresponding events. (This may or may not be a single valid key sequence, depending on what events you use; see Section 23.1 [Key Sequences], page 481.) If *need-vector* is non-`nil`, the return value is always a vector.

25.6 Help Functions

Emacs provides a variety of built-in help functions, all accessible to the user as subcommands of the prefix `C-h`. For more information about them, see Section "Help" in *The GNU Emacs Manual*. Here we describe some program-level interfaces to the same information.

`apropos` *pattern* **&optional** *do-all* [Command]

This function finds all meaningful symbols whose names contain a match for the apropos pattern *pattern*. An apropos pattern is either a word to match, a space-separated list of words of which at least two must match, or a regular expression (if any special regular expression characters occur). A symbol is meaningful if it has a definition as a function, variable, or face, or has properties.

The function returns a list of elements that look like this:

```
(symbol score function-doc variable-doc
 plist-doc widget-doc face-doc group-doc)
```

Here, *score* is an integer measure of how important the symbol seems to be as a match. Each of the remaining elements is a documentation string, or `nil`, for *symbol* as a function, variable, etc.

It also displays the symbols in a buffer named `*Apropos*`, each with a one-line description taken from the beginning of its documentation string.

If *do-all* is non-`nil`, or if the user option `apropos-do-all` is non-`nil`, then `apropos` also shows key bindings for the functions that are found; it also shows *all* interned symbols, not just meaningful ones (and it lists them in the return value as well).

`help-map` [Variable]
The value of this variable is a local keymap for characters following the Help key, `C-h`.

`help-command` [Prefix Command]
This symbol is not a function; its function definition cell holds the keymap known as `help-map`. It is defined in `help.el` as follows:

```
(keymap-set global-map (key-description (string help-char)) 'help-command)
(fset 'help-command help-map)
```

`help-char` [User Option]
The value of this variable is the help character—the character that Emacs recognizes as meaning Help. By default, its value is 8, which stands for `C-h`. When Emacs reads this character, if `help-form` is a non-`nil` Lisp expression, it evaluates that expression, and displays the result in a window if it is a string.

Usually the value of `help-form` is `nil`. Then the help character has no special meaning at the level of command input, and it becomes part of a key sequence in the normal way. The standard key binding of `C-h` is a prefix key for several general-purpose help features.

The help character is special after prefix keys, too. If it has no binding as a subcommand of the prefix key, it runs `describe-prefix-bindings`, which displays a list of all the subcommands of the prefix key.

`help-event-list` [User Option]
The value of this variable is a list of event types that serve as alternative help characters. These events are handled just like the event specified by `help-char`.

`help-form` [Variable]
If this variable is non-`nil`, its value is a form to evaluate whenever the character `help-char` is read. If evaluating the form produces a string, that string is displayed.

A command that calls `read-event`, `read-char-choice`, `read-char`, `read-char-from-minibuffer`, or `y-or-n-p` probably should bind `help-form` to a non-`nil` expression while it does input. (The time when you should not do this is when `C-h` has some other meaning.) Evaluating this expression should result in a string that explains what the input is for and how to enter it properly.

Entry to the minibuffer binds this variable to the value of `minibuffer-help-form` (see [Definition of minibuffer-help-form], page 423).

`prefix-help-command` [Variable]
This variable holds a function to print help for a prefix key. The function is called when the user types a prefix key followed by the help character, and the help character has no binding after that prefix. The variable's default value is `describe-prefix-bindings`.

`describe-prefix-bindings` [Command]
This function calls `describe-bindings` to display a list of all the subcommands of the prefix key of the most recent key sequence. The prefix described consists of all but the last event of that key sequence. (The last event is, presumably, the help character.)

The following two functions are meant for modes that want to provide help without relinquishing control, such as the electric modes. Their names begin with '`Helper`' to distinguish them from the ordinary help functions.

`Helper-describe-bindings` [Command]
: This command pops up a window displaying a help buffer containing a listing of all of the key bindings from both the local and global keymaps. It works by calling `describe-bindings`.

`Helper-help` [Command]
: This command provides help for the current mode. It prompts the user in the minibuffer with the message '`Help (Type ? for further options)`', and then provides assistance in finding out what the key bindings are, and what the mode is intended for. It returns `nil`.

  This can be customized by changing the map `Helper-help-map`.

`data-directory` [Variable]
: This variable holds the name of the directory in which Emacs finds certain documentation and text files that come with Emacs.

`help-buffer` [Function]
: This function returns the name of the help buffer, which is normally `*Help*`; if such a buffer does not exist, it is first created.

`with-help-window` *buffer-or-name body...* [Macro]
: This macro evaluates *body* like `with-output-to-temp-buffer` (see Section 41.8 [Temporary Displays], page 1150), inserting any output produced by its forms into a buffer specified by *buffer-or-name*, which can be a buffer or the name of a buffer. (Frequently, *buffer-or-name* is the value returned by the function `help-buffer`.) This macro puts the specified buffer into Help mode and displays a message telling the user how to quit and scroll the help window. It selects the help window if the current value of the user option `help-window-select` has been set accordingly. It returns the last value in *body*.

`help-setup-xref` *item interactive-p* [Function]
: This function updates the cross reference data in the `*Help*` buffer, which is used to regenerate the help information when the user clicks on the '`Back`' or '`Forward`' buttons. Most commands that use the `*Help*` buffer should invoke this function before clearing the buffer. The *item* argument should have the form `(`*`function`*` . `*`args`*`)`, where *function* is a function to call, with argument list *args*, to regenerate the help buffer. The *interactive-p* argument is non-`nil` if the calling command was invoked interactively; in that case, the stack of items for the `*Help*` buffer's '`Back`' buttons is cleared.

See [describe-symbols example], page 599, for an example of using `help-buffer`, `with-help-window`, and `help-setup-xref`.

`make-help-screen` *fname help-line help-text help-map* [Macro]
: This macro defines a help command named *fname* that acts like a prefix key that shows a list of the subcommands it offers.

When invoked, *fname* displays *help-text* in a window, then reads and executes a key sequence according to *help-map*. The string *help-text* should describe the bindings available in *help-map*.

The command *fname* is defined to handle a few events itself, by scrolling the display of *help-text*. When *fname* reads one of those special events, it does the scrolling and then reads another event. When it reads an event that is not one of those few, and which has a binding in *help-map*, it executes that key's binding and then returns.

The argument *help-line* should be a single-line summary of the alternatives in *help-map*. In the current version of Emacs, this argument is used only if you set the option `three-step-help` to `t`.

This macro is used in the command `help-for-help` which is the binding of *`C-h C-h`*.

`three-step-help` [User Option]
If this variable is non-`nil`, commands defined with `make-help-screen` display their *help-line* strings in the echo area at first, and display the longer *help-text* strings only if the user types the help character again.

25.7 Documentation Groups

Emacs can list functions based on various groupings. For instance, `string-trim` and `mapconcat` are "string" functions, so *`M-x shortdoc RET string RET`* will give an overview of functions that operate on strings.

The documentation groups are created with the `define-short-documentation-group` macro.

`define-short-documentation-group` *group* **&rest** *functions* [Macro]
Define *group* as a group of functions, and provide short summaries of using those functions. The optional argument *functions* is a list whose elements are of the form:

```
(func [keyword val]...)
```

The following keywords are recognized:

`:eval` The value should be a form that has no side effect when evaluated. The form will be used in the documentation by printing it with `prin1` (see Section 20.5 [Output Functions], page 378). However, if the form is a string, it will be inserted as-is, and the string will then be `read` to yield the form. In any case, the form will then be evaluated, and the result used. For instance:

```
:eval (concat "foo" "bar" "zot")
:eval "(make-string 5 ?x)"
```

will result in:

```
(concat "foo" "bar" "zot")
⇒ "foobarzot"
(make-string 5 ?x)
⇒ "xxxxx"
```

(The reason for allowing both Lisp forms and strings here is so that printing could be controlled in the few cases where a certain presentation

of the form is wished for. In the example, '`?x`' would otherwise have been printed as '`120`' if it hadn't been included in a string.)

`:no-eval`
: This is like `:eval`, except that the form will not be evaluated. In these cases, a `:result` element of some kind (see below) should be included.

```
:no-eval (file-symlink-p "/tmp/foo")
:eg-result t
```

`:no-eval*`
: Like `:no-eval`, but always inserts '`[it depends]`' as the result. For instance:

```
:no-eval* (buffer-string)
```

will result in:

```
(buffer-string)
→ [it depends]
```

`:no-value`
: Like `:no-eval`, but is used when the function in question has no well-defined return value, and is used for side effect only.

`:result`
: Used to output the result from non-evaluating example forms.

```
:no-eval (setcar list 'c)
:result c
```

`:eg-result`
: Used to output an example result from non-evaluating example forms. For instance:

```
:no-eval (looking-at "f[0-9]")
:eg-result t
```

will result in:

```
(looking-at "f[0-9]")
eg. → t
```

`:result-string`
`:eg-result-string`
: These two are the same as `:result` and `:eg-result`, respectively, but are inserted as is. This is useful when the result is unreadable or should be of a particular form:

```
:no-eval (find-file "/tmp/foo")
:eg-result-string "#<buffer foo>"
:no-eval (default-file-modes)
:eg-result-string "#o755"
```

`:no-manual`
: Indicates that this function is not documented in the manual.

`:args`
: By default, the function's actual argument list is shown. If `:args` is present, they are used instead.

```
:args (regexp string)
```

Here's a very short example:

```
(define-short-documentation-group string
  "Creating Strings"
  (substring
   :eval (substring "foobar" 0 3)
   :eval (substring "foobar" 3))
  (concat
   :eval (concat "foo" "bar" "zot")))
```

The first argument is the name of the group to be defined, and then follows any number of function descriptions.

A function can belong to any number of documentation groups.

In addition to function descriptions, the list can also have string elements, which are used to divide a documentation group into sections.

`shortdoc-add-function` *group section elem* [Function]

Lisp packages can add functions to groups with this function. Each *elem* should be a function description, as described above. *group* is the function group, and *section* is what section in the function group to insert the function into.

If *group* doesn't exist, it will be created. If *section* doesn't exist, it will be added to the end of the function group.

26 Files

This chapter describes the Emacs Lisp functions and variables to find, create, view, save, and otherwise work with files and directories. A few other file-related functions are described in Chapter 28 [Buffers], page 673, and those related to backups and auto-saving are described in Chapter 27 [Backups and Auto-Saving], page 661.

Many of the file functions take one or more arguments that are file names. A file name is a string. Most of these functions expand file name arguments using the function `expand-file-name`, so that `~` is handled correctly, as are relative file names (including `../` and the empty string). See Section 26.9.4 [File Name Expansion], page 641.

In addition, certain *magic* file names are handled specially. For example, when a remote file name is specified, Emacs accesses the file over the network via an appropriate protocol. See Section "Remote Files" in *The GNU Emacs Manual*. This handling is done at a very low level, so you may assume that all the functions described in this chapter accept magic file names as file name arguments, except where noted. See Section 26.12 [Magic File Names], page 651, for details.

When file I/O functions signal Lisp errors, they usually use the condition `file-error` (see Section 11.7.3.3 [Handling Errors], page 182). The error message is in most cases obtained from the operating system, according to locale `system-messages-locale`, and decoded using coding system `locale-coding-system` (see Section 34.12 [Locales], page 995).

26.1 Visiting Files

Visiting a file means reading a file into a buffer. Once this is done, we say that the buffer is *visiting* that file, and call the file *the visited file* of the buffer.

A file and a buffer are two different things. A file is information recorded permanently in the computer (unless you delete it). A buffer, on the other hand, is information inside of Emacs that will vanish at the end of the editing session (or when you kill the buffer). When a buffer is visiting a file, it contains information copied from the file. The copy in the buffer is what you modify with editing commands. Changes to the buffer do not change the file; to make the changes permanent, you must *save* the buffer, which means copying the altered buffer contents back into the file.

Despite the distinction between files and buffers, people often refer to a file when they mean a buffer and vice-versa. Indeed, we say, "I am editing a file", rather than, "I am editing a buffer that I will soon save as a file of the same name". Humans do not usually need to make the distinction explicit. When dealing with a computer program, however, it is good to keep the distinction in mind.

26.1.1 Functions for Visiting Files

This section describes the functions normally used to visit files. For historical reasons, these functions have names starting with '`find-`' rather than '`visit-`'. See Section 28.4 [Buffer File Name], page 677, for functions and variables that access the visited file name of a buffer or that find an existing buffer by its visited file name.

In a Lisp program, if you want to look at the contents of a file but not alter it, the fastest way is to use `insert-file-contents` in a temporary buffer. Visiting the file is not necessary and takes longer. See Section 26.3 [Reading from Files], page 616.

`find-file` *filename* **&optional** *wildcards* [Command]
This command selects a buffer visiting the file *filename*, using an existing buffer if there is one, and otherwise creating a new buffer and reading the file into it. It also returns that buffer.

Aside from some technical details, the body of the `find-file` function is basically equivalent to:

```
(switch-to-buffer (find-file-noselect filename nil nil wildcards))
```

(See `switch-to-buffer` in Section 29.12 [Switching Buffers], page 723.)

If *wildcards* is non-`nil`, which is always true in an interactive call, then `find-file` expands wildcard characters in *filename* and visits all the matching files.

When `find-file` is called interactively, it prompts for *filename* in the minibuffer.

`find-file-literally` *filename* [Command]
This command visits *filename*, like `find-file` does, but it does not perform any format conversions (see Section 26.13 [Format Conversion], page 656), character code conversions (see Section 34.10 [Coding Systems], page 980), or end-of-line conversions (see Section 34.10.1 [Coding System Basics], page 980). The buffer visiting the file is made unibyte, and its major mode is Fundamental mode, regardless of the file name. File local variable specifications in the file (see Section 12.12 [File Local Variables], page 213) are ignored, and automatic decompression and adding a newline at the end of the file due to `require-final-newline` (see Section 26.2 [Saving Buffers], page 614) are also disabled.

Note that if Emacs already has a buffer visiting the same file non-literally, it will not visit the same file literally, but instead just switch to the existing buffer. If you want to be sure of accessing a file's contents literally, you should create a temporary buffer and then read the file contents into it using `insert-file-contents-literally` (see Section 26.3 [Reading from Files], page 616).

`find-file-noselect` *filename* **&optional** *nowarn rawfile wildcards* [Function]
This function is the guts of all the file-visiting functions. It returns a buffer visiting the file *filename*. You may make the buffer current or display it in a window if you wish, but this function does not do so.

The function returns an existing buffer if there is one; otherwise it creates a new buffer and reads the file into it. When `find-file-noselect` uses an existing buffer, it first verifies that the file has not changed since it was last visited or saved in that buffer. If the file has changed, this function asks the user whether to reread the changed file. If the user says '`yes`', any edits previously made in the buffer are lost.

Reading the file involves decoding the file's contents (see Section 34.10 [Coding Systems], page 980), including end-of-line conversion, and format conversion (see Section 26.13 [Format Conversion], page 656). If *wildcards* is non-`nil`, then `find-file-noselect` expands wildcard characters in *filename* and visits all the matching files.

This function displays warning or advisory messages in various peculiar cases, unless the optional argument *nowarn* is non-`nil`. For example, if it needs to create a buffer, and there is no file named *filename*, it displays the message '`(New file)`' in the echo area, and leaves the buffer empty.

The `find-file-noselect` function normally calls `after-find-file` after reading the file (see Section 26.1.2 [Subroutines of Visiting], page 613). That function sets the buffer major mode, parses local variables, warns the user if there exists an auto-save file more recent than the file just visited, and finishes by running the functions in `find-file-hook`.

If the optional argument *rawfile* is non-`nil`, then `after-find-file` is not called, and the `find-file-not-found-functions` are not run in case of failure. What's more, a non-`nil` *rawfile* value suppresses coding system conversion and format conversion.

The `find-file-noselect` function usually returns the buffer that is visiting the file *filename*. But, if wildcards are actually used and expanded, it returns a list of buffers that are visiting the various files.

```
(find-file-noselect "/etc/fstab")
     ⇒ #<buffer fstab>
```

`find-file-other-window` *filename* **&optional** *wildcards* [Command]
This command selects a buffer visiting the file *filename*, but does so in a window other than the selected window. It may use another existing window or split a window; see Section 29.12 [Switching Buffers], page 723.

When this command is called interactively, it prompts for *filename*.

`find-file-read-only` *filename* **&optional** *wildcards* [Command]
This command selects a buffer visiting the file *filename*, like `find-file`, but it marks the buffer as read-only. See Section 28.7 [Read Only Buffers], page 682, for related functions and variables.

When this command is called interactively, it prompts for *filename*.

`find-file-wildcards` [User Option]
If this variable is non-`nil`, then the various `find-file` commands check for wildcard characters and visit all the files that match them (when invoked interactively or when their *wildcards* argument is non-`nil`). If this option is `nil`, then the `find-file` commands ignore their *wildcards* argument and never treat wildcard characters specially.

`find-file-hook` [User Option]
The value of this variable is a list of functions to be called after a file is visited. The file's local-variables specification (if any) will have been processed before the hooks are run. The buffer visiting the file is current when the hook functions are run.

This variable is a normal hook. See Section 24.1 [Hooks], page 525.

`find-file-not-found-functions` [Variable]
The value of this variable is a list of functions to be called when `find-file` or `find-file-noselect` is passed a nonexistent file name. `find-file-noselect` calls these functions as soon as it detects a nonexistent file. It calls them in the order of the list, until one of them returns non-`nil`. `buffer-file-name` is already set up.

This is not a normal hook because the values of the functions are used, and in many cases only some of the functions are called.

`find-file-literally` [Variable]

This buffer-local variable, if set to a non-`nil` value, makes `save-buffer` behave as if the buffer were visiting its file literally, i.e., without conversions of any kind. The command `find-file-literally` sets this variable's local value, but other equivalent functions and commands can do that as well, e.g., to avoid automatic addition of a newline at the end of the file. This variable is permanent local, so it is unaffected by changes of major modes.

26.1.2 Subroutines of Visiting

The `find-file-noselect` function uses two important subroutines which are sometimes useful in user Lisp code: `create-file-buffer` and `after-find-file`. This section explains how to use them.

`create-file-buffer` *filename* [Function]

This function creates a suitably named buffer for visiting *filename*, and returns it. It uses *filename* (sans directory) as the name if that name is free; otherwise, it appends a string such as '`<2>`' to get an unused name. See also Section 28.9 [Creating Buffers], page 686. Note that the `uniquify` library affects the result of this function. See Section "Uniquify" in *The GNU Emacs Manual*.

Please note: `create-file-buffer` does *not* associate the new buffer with a file and does not select the buffer. It also does not use the default major mode.

```
(create-file-buffer "foo")
     ⇒ #<buffer foo>
(create-file-buffer "foo")
     ⇒ #<buffer foo<2>>
(create-file-buffer "foo")
     ⇒ #<buffer foo<3>>
```

This function is used by `find-file-noselect`. It uses `generate-new-buffer` (see Section 28.9 [Creating Buffers], page 686).

`after-find-file` **&optional** *error warn noauto* [Function]
after-find-file-from-revert-buffer nomodes

This function sets the buffer major mode, and parses local variables (see Section 24.2.2 [Auto Major Mode], page 532). It is called by `find-file-noselect` and by the default revert function (see Section 27.3 [Reverting], page 669).

If reading the file got an error because the file does not exist, but its directory does exist, the caller should pass a non-`nil` value for *error*. In that case, `after-find-file` issues a warning: '`(New file)`'. For more serious errors, the caller should usually not call `after-find-file`.

If *warn* is non-`nil`, then this function issues a warning if an auto-save file exists and is more recent than the visited file.

If *noauto* is non-`nil`, that says not to enable or disable Auto-Save mode. The mode remains enabled if it was enabled before.

If *after-find-file-from-revert-buffer* is non-`nil`, that means this call was from `revert-buffer`. This has no direct effect, but some mode functions and hook functions check the value of this variable.

If *nomodes* is non-`nil`, that means don't alter the buffer's major mode, don't process local variables specifications in the file, and don't run `find-file-hook`. This feature is used by `revert-buffer` in some cases.

The last thing `after-find-file` does is call all the functions in the list `find-file-hook`.

26.2 Saving Buffers

When you edit a file in Emacs, you are actually working on a buffer that is visiting that file—that is, the contents of the file are copied into the buffer and the copy is what you edit. Changes to the buffer do not change the file until you *save* the buffer, which means copying the contents of the buffer into the file. Buffers which are not visiting a file can still be "saved", in a sense, using functions in the buffer-local `write-contents-functions` hook.

`save-buffer` **&optional** *backup-option* [Command]

This function saves the contents of the current buffer in its visited file if the buffer has been modified since it was last visited or saved. Otherwise it does nothing.

`save-buffer` is responsible for making backup files. Normally, *backup-option* is `nil`, and `save-buffer` makes a backup file only if this is the first save since visiting the file. Other values for *backup-option* request the making of backup files in other circumstances:

- With an argument of 4 or 64, reflecting 1 or 3 `C-u`'s, the `save-buffer` function marks this version of the file to be backed up when the buffer is next saved.
- With an argument of 16 or 64, reflecting 2 or 3 `C-u`'s, the `save-buffer` function unconditionally backs up the previous version of the file before saving it.
- With an argument of 0, unconditionally do *not* make any backup file.

`save-some-buffers` **&optional** *save-silently-p pred* [Command]

This command saves some modified file-visiting buffers. Normally it asks the user about each buffer. But if *save-silently-p* is non-`nil`, it saves all the file-visiting buffers without querying the user.

The optional *pred* argument provides a predicate that controls which buffers to ask about (or to save silently if *save-silently-p* is non-`nil`). If *pred* is `nil`, that means to use the value of `save-some-buffers-default-predicate` instead of *pred*. If the result is `nil`, it means ask only about file-visiting buffers. If it is `t`, that means also offer to save certain other non-file buffers—those that have a non-`nil` buffer-local value of `buffer-offer-save` (see Section 28.10 [Killing Buffers], page 687). A user who says '`yes`' to saving a non-file buffer is asked to specify the file name to use. The `save-buffers-kill-emacs` function passes the value `t` for *pred*.

If the predicate is neither `t` nor `nil`, then it should be a function of no arguments. It will be called in each buffer to decide whether to offer to save that buffer. If it returns a non-`nil` value in a certain buffer, that means do offer to save that buffer.

`write-file` *filename* **&optional** *confirm* [Command]

This function writes the current buffer into file *filename*, makes the buffer visit that file, and marks it not modified. Then it renames the buffer based on *filename*, appending a string like '`<2>`' if necessary to make a unique buffer name. It does most

of this work by calling `set-visited-file-name` (see Section 28.4 [Buffer File Name], page 677) and `save-buffer`.

If *confirm* is non-`nil`, that means to ask for confirmation before overwriting an existing file. Interactively, confirmation is required, unless the user supplies a prefix argument.

If *filename* is a directory name (see Section 26.9.3 [Directory Names], page 640), `write-file` uses the name of the visited file, in directory *filename*. If the buffer is not visiting a file, it uses the buffer name instead.

Saving a buffer runs several hooks. It also performs format conversion (see Section 26.13 [Format Conversion], page 656). Note that these hooks, described below, are only run by `save-buffer`, they are not run by other primitives and functions that write buffer text to files, and in particular auto-saving (see Section 27.2 [Auto-Saving], page 666) doesn't run these hooks.

`write-file-functions` [Variable]

The value of this variable is a list of functions to be called before writing out a buffer to its visited file. If one of them returns non-`nil`, the file is considered already written and the rest of the functions are not called, nor is the usual code for writing the file executed.

If a function in `write-file-functions` returns non-`nil`, it is responsible for making a backup file (if that is appropriate). To do so, execute the following code:

```
(or buffer-backed-up (backup-buffer))
```

You might wish to save the file modes value returned by `backup-buffer` and use that (if non-`nil`) to set the mode bits of the file that you write. This is what `save-buffer` normally does. See Section 27.1.1 [Making Backup Files], page 661.

The hook functions in `write-file-functions` are also responsible for encoding the data (if desired): they must choose a suitable coding system and end-of-line conversion (see Section 34.10.3 [Lisp and Coding Systems], page 982), perform the encoding (see Section 34.10.7 [Explicit Encoding], page 991), and set `last-coding-system-used` to the coding system that was used (see Section 34.10.2 [Encoding and I/O], page 981).

If you set this hook locally in a buffer, it is assumed to be associated with the file or the way the contents of the buffer were obtained. Thus the variable is marked as a permanent local, so that changing the major mode does not alter a buffer-local value. On the other hand, calling `set-visited-file-name` will reset it. If this is not what you want, you might like to use `write-contents-functions` instead.

Even though this is not a normal hook, you can use `add-hook` and `remove-hook` to manipulate the list. See Section 24.1 [Hooks], page 525.

`write-contents-functions` [Variable]

This works just like `write-file-functions`, but it is intended for hooks that pertain to the buffer's contents, not to the particular visited file or its location, and can be used to create arbitrary save processes for buffers that aren't visiting files at all. Such hooks are usually set up by major modes, as buffer-local bindings for this variable. This variable automatically becomes buffer-local whenever it is set; switching to a new major mode always resets this variable, but calling `set-visited-file-name` does not.

If any of the functions in this hook returns non-`nil`, the file is considered already written and the rest are not called and neither are the functions in `write-file-functions`.

When using this hook to save buffers that are not visiting files (for instance, special-mode buffers), keep in mind that, if the function fails to save correctly and returns a `nil` value, `save-buffer` will go on to prompt the user for a file to save the buffer in. If this is undesirable, consider having the function fail by raising an error.

`before-save-hook` [User Option]
This normal hook runs before a buffer is saved in its visited file, regardless of whether that is done normally or by one of the hooks described above. For instance, the `copyright.el` program uses this hook to make sure the file you are saving has the current year in its copyright notice.

`after-save-hook` [User Option]
This normal hook runs after a buffer has been saved in its visited file.

`file-precious-flag` [User Option]
If this variable is non-`nil`, then `save-buffer` protects against I/O errors while saving by writing the new file to a temporary name instead of the name it is supposed to have, and then renaming it to the intended name after it is clear there are no errors. This procedure prevents problems such as a lack of disk space from resulting in an invalid file.

As a side effect, backups are necessarily made by copying. See Section 27.1.2 [Rename or Copy], page 663. Yet, at the same time, saving a precious file always breaks all hard links between the file you save and other file names.

Some modes give this variable a non-`nil` buffer-local value in particular buffers.

`require-final-newline` [User Option]
This variable determines whether files may be written out that do *not* end with a newline. If the value of the variable is `t`, then `save-buffer` silently adds a newline at the end of the buffer whenever it does not already end in one. If the value is `visit`, Emacs adds a missing newline just after it visits the file. If the value is `visit-save`, Emacs adds a missing newline both on visiting and on saving. For any other non-`nil` value, `save-buffer` asks the user whether to add a newline each time the case arises.

If the value of the variable is `nil`, then `save-buffer` doesn't add newlines at all. `nil` is the default value, but a few major modes set it to `t` in particular buffers.

See also the function `set-visited-file-name` (see Section 28.4 [Buffer File Name], page 677).

26.3 Reading from Files

To copy the contents of a file into a buffer, use the function `insert-file-contents`. (Don't use the command `insert-file` in a Lisp program, as that sets the mark.)

`insert-file-contents` *filename* **&optional** *visit beg end replace* [Function]
This function inserts the contents of file *filename* into the current buffer after point. It returns a list of the absolute file name and the length of the data inserted. An error is signaled if *filename* is not the name of a file that can be read.

This function checks the file contents against the defined file formats, and converts the file contents if appropriate and also calls the functions in the list `after-insert-file-functions`. See Section 26.13 [Format Conversion], page 656. Normally, one of the functions in the `after-insert-file-functions` list determines the coding system (see Section 34.10 [Coding Systems], page 980) used for decoding the file's contents, including end-of-line conversion. However, if the file contains null bytes, it is by default visited without any code conversions. See Section 34.10.3 [Lisp and Coding Systems], page 982.

If *visit* is non-`nil`, this function additionally marks the buffer as unmodified and sets up various fields in the buffer so that it is visiting the file *filename*: these include the buffer's visited file name and its last save file modtime. This feature is used by `find-file-noselect` and you probably should not use it yourself.

If *beg* and *end* are non-`nil`, they should be numbers that are byte offsets specifying the portion of the file to insert. In this case, *visit* must be `nil`. For example,

```
(insert-file-contents filename nil 0 500)
```

inserts the characters coded by the first 500 bytes of a file.

If *beg* or *end* happens to be in the middle of a character's multibyte sequence, Emacs's character code conversion will insert one or more eight-bit characters (a.k.a. "raw bytes") (see Section 34.7 [Character Sets], page 976) into the buffer. If you want to read part of a file this way, we recommend to bind `coding-system-for-read` to a suitable value around the call to this function (see Section 34.10.6 [Specifying Coding Systems], page 989), and to write Lisp code which will check for raw bytes at the boundaries, read the entire sequence of these bytes, and convert them back to valid characters.

If the argument *replace* is non-`nil`, it means to replace the contents of the buffer (actually, just the accessible portion) with the contents of the file. This is better than simply deleting the buffer contents and inserting the whole file, because (1) it preserves some marker positions and (2) it puts less data in the undo list.

It is possible to read a special file (such as a FIFO or an I/O device) with `insert-file-contents`, as long as *replace*, and *visit* and *beg* are `nil`. However, you should normally use an *end* argument for these files to avoid inserting (potentially) unlimited data into the buffer (for instance, when inserting data from `/dev/urandom`).

`insert-file-contents-literally` *filename* **&optional** *visit beg end replace* [Function]

This function works like `insert-file-contents` except that each byte in the file is handled separately, being converted into an eight-bit character if needed. It does not run `after-insert-file-functions`, and does not do format decoding, character code conversion, automatic uncompression, and so on.

If you want to pass a file name to another process so that another program can read the file, use the function `file-local-copy`; see Section 26.12 [Magic File Names], page 651.

26.4 Writing to Files

You can write the contents of a buffer, or part of a buffer, directly to a file on disk using the `append-to-file` and `write-region` functions. Don't use these functions to write to files that are being visited; that could cause confusion in the mechanisms for visiting.

`append-to-file` *start end filename* [Command]

This function appends the contents of the region delimited by *start* and *end* in the current buffer to the end of file *filename*. If that file does not exist, it is created. This function returns `nil`.

An error is signaled if you cannot write or create *filename*.

When called from Lisp, this function is completely equivalent to:

```
(write-region start end filename t)
```

`write-region` *start end filename* **&optional** *append visit lockname mustbenew* [Command]

This function writes the region delimited by *start* and *end* in the current buffer into the file specified by *filename*.

If *start* is `nil`, then the command writes the entire buffer contents (*not* just the accessible portion) to the file and ignores *end*.

If *start* is a string, then `write-region` writes or appends that string, rather than text from the buffer. *end* is ignored in this case.

If *append* is non-`nil`, then the specified text is appended to the existing file contents (if any). If *append* is a number, `write-region` seeks to that byte offset from the start of the file and writes the data from there.

If *mustbenew* is non-`nil`, then `write-region` asks for confirmation if *filename* names an existing file. If *mustbenew* is the symbol `excl`, then `write-region` does not ask for confirmation, but instead it signals an error `file-already-exists` if the file already exists. Although `write-region` normally follows a symbolic link and creates the pointed-to file if the symbolic link is dangling, it does not follow symbolic links if *mustbenew* is `excl`.

The test for an existing file, when *mustbenew* is `excl`, uses a special system feature. At least for files on a local disk, there is no chance that some other program could create a file of the same name before Emacs does, without Emacs's noticing.

If *visit* is `t`, then Emacs establishes an association between the buffer and the file: the buffer is then visiting that file. It also sets the last file modification time for the current buffer to *filename*'s modtime, and marks the buffer as not modified. This feature is used by `save-buffer`, but you probably should not use it yourself.

If *visit* is a string, it specifies the file name to visit. This way, you can write the data to one file (*filename*) while recording the buffer as visiting another file (*visit*). The argument *visit* is used in the echo area message and also for file locking; *visit* is stored in `buffer-file-name`. This feature is used to implement `file-precious-flag`; don't use it yourself unless you really know what you're doing.

The optional argument *lockname*, if non-`nil`, specifies the file name to use for purposes of locking and unlocking, overriding *filename* and *visit* for that purpose.

The function `write-region` converts the data which it writes to the appropriate file formats specified by `buffer-file-format` and also calls the functions in the list `write-region-annotate-functions`. See Section 26.13 [Format Conversion], page 656.

Normally, `write-region` displays the message '`Wrote` *filename*' in the echo area. This message is inhibited if *visit* is neither `t` nor `nil` nor a string, or if Emacs is operating in batch mode (see Section 42.17 [Batch Mode], page 1294). This feature is useful for programs that use files for internal purposes, files that the user does not need to know about.

`write-region-inhibit-fsync` [Variable]

If this variable's value is `nil`, `write-region` uses the `fsync` system call after writing a file. Although this slows Emacs down, it lessens the risk of data loss after power failure. If the value is `t`, Emacs does not use `fsync`. The default value is `nil` when Emacs is interactive, and `t` when Emacs runs in batch mode. See Section 26.8 [Files and Storage], page 636.

`with-temp-file` *file body...* [Macro]

The `with-temp-file` macro evaluates the *body* forms with a temporary buffer as the current buffer; then, at the end, it writes the buffer contents into file *file*. It kills the temporary buffer when finished, restoring the buffer that was current before the `with-temp-file` form. Then it returns the value of the last form in *body*.

The current buffer is restored even in case of an abnormal exit via `throw` or error (see Section 11.7 [Nonlocal Exits], page 177).

Like `with-temp-buffer` (see [Current Buffer], page 675), the temporary buffer used by this macro does not run the hooks `kill-buffer-hook`, `kill-buffer-query-functions` (see Section 28.10 [Killing Buffers], page 687), and `buffer-list-update-hook` (see Section 28.8 [Buffer List], page 683).

26.5 File Locks

When two users edit the same file at the same time, they are likely to interfere with each other. Emacs tries to prevent this situation from arising by recording a *file lock* when a file is being modified. Emacs can then detect the first attempt to modify a buffer visiting a file that is locked by another Emacs job, and ask the user what to do. The file lock is really a file, a symbolic link with a special name, stored in the same directory as the file you are editing. The name is constructed by prepending `.#` to the file name of the buffer. The target of the symbolic link will be of the form *`user@host.pid:boot`*, where *user* is replaced with the current username (from `user-login-name`), *host* with the name of the host where Emacs is running (from `system-name`), *pid* with Emacs's process id, and *boot* with the time since the last reboot. *`:boot`* is omitted if the boot time is unavailable. (On file systems that do not support symbolic links, a regular file is used instead, with contents of the form *`user@host.pid:boot`*.)

When you access files using NFS, there may be a small probability that you and another user will both lock the same file simultaneously. If this happens, it is possible for the two users to make changes simultaneously, but Emacs will still warn the user who saves second.

Also, the detection of modification of a buffer visiting a file changed on disk catches some cases of simultaneous editing; see Section 28.6 [Modification Time], page 680.

`file-locked-p` *filename* [Function]
: This function returns `nil` if the file *filename* is not locked. It returns `t` if it is locked by this Emacs process, and it returns the name of the user who has locked it if it is locked by some other job.

```
(file-locked-p "foo")
     ⇒ nil
```

`lock-buffer` **&optional** *filename* [Function]
: This function locks the file *filename*, if the current buffer is modified. The argument *filename* defaults to the current buffer's visited file. Nothing is done if the current buffer is not visiting a file, or is not modified, or if the option `create-lockfiles` is `nil`.

`unlock-buffer` [Function]
: This function unlocks the file being visited in the current buffer, if the buffer is modified. If the buffer is not modified, then the file should not be locked, so this function does nothing. It also does nothing if the current buffer is not visiting a file, or is not locked. This function handles file system errors by calling `display-warning` and otherwise ignores the error.

`create-lockfiles` [User Option]
: If this variable is `nil`, Emacs does not lock files.

`lock-file-name-transforms` [User Option]
: By default, Emacs creates the lock files in the same directory as the files that are being locked. This can be changed by customizing this variable. Is has the same syntax as `auto-save-file-name-transforms` (see Section 27.2 [Auto-Saving], page 666). For instance, to make Emacs write all the lock files to `/var/tmp/`, you could say something like:

```
(setq lock-file-name-transforms
      '(("\\`/.*/\\([^/]+\\)\\'" "/var/tmp/\\1" t)))
```

`ask-user-about-lock` *file other-user* [Function]
: This function is called when the user tries to modify *file*, but it is locked by another user named *other-user*. The default definition of this function asks the user to say what to do. The value this function returns determines what Emacs does next:

- A value of `t` says to grab the lock on the file. Then this user may edit the file and *other-user* loses the lock.
- A value of `nil` says to ignore the lock and let this user edit the file anyway.
- This function may instead signal a `file-locked` error, in which case the change that the user was about to make does not take place.

  The error message for this error looks like this:

  error `File is locked: file other-user`

  where `file` is the name of the file and *other-user* is the name of the user who has locked the file.

If you wish, you can replace the `ask-user-about-lock` function with your own version that makes the decision in another way.

`remote-file-name-inhibit-locks` [User Option]
You can prevent the creation of remote lock files by setting the variable `remote-file-name-inhibit-locks` to `t`.

`lock-file-mode` [Command]
This command, called interactively, toggles the local value of `create-lockfiles` in the current buffer.

26.6 Information about Files

This section describes the functions for retrieving various types of information about files (or directories or symbolic links), such as whether a file is readable or writable, and its size. These functions all take arguments which are file names. Except where noted, these arguments need to specify existing files, or an error is signaled.

Be careful with file names that end in spaces. On some filesystems (notably, MS-Windows), trailing whitespace characters in file names are silently and automatically ignored.

26.6.1 Testing Accessibility

These functions test for permission to access a file for reading, writing, or execution. Unless explicitly stated otherwise, they follow symbolic links. See Section 26.6.2 [Kinds of Files], page 623.

On some operating systems, more complex sets of access permissions can be specified, via mechanisms such as Access Control Lists (ACLs). See Section 26.6.5 [Extended Attributes], page 629, for how to query and set those permissions.

`file-exists-p` *filename* [Function]
This function returns `t` if a file named *filename* appears to exist. This does not mean you can necessarily read the file, only that you can probably find out its attributes. (On GNU and other POSIX-like systems, this is true if the file exists and you have execute permission on the containing directories, regardless of the permissions of the file itself.)

If the file does not exist, or if there was trouble determining whether the file exists, this function returns `nil`.

Since a file name that is an empty string is interpreted relative to the current buffer's default directory (see Section 26.9.2 [Relative File Names], page 639), calling `file-exists-p` with an argument that is an empty string will report about the buffer's default directory.

Directories are files, so `file-exists-p` can return `t` when given a directory. However, because `file-exists-p` follows symbolic links, it returns `t` for a symbolic link name only if the target of the link exists; if your Lisp program needs to consider *dangling symlinks* whose target doesn't exist as existing files, use `file-attributes` (see Section 26.6.4 [File Attributes], page 626) instead of `file-exists-p`.

`file-readable-p` *filename* [Function]
: This function returns `t` if a file named *filename* exists and you can read it. It returns `nil` otherwise.

`file-executable-p` *filename* [Function]
: This function returns `t` if a file named *filename* exists and you can execute it. It returns `nil` otherwise. On GNU and other POSIX-like systems, if the file is a directory, execute permission means you can check the existence and attributes of files inside the directory, and open those files if their modes permit.

`file-writable-p` *filename* [Function]
: This function returns `t` if the file *filename* can be written or created by you, and `nil` otherwise. A file is writable if the file exists and you can write it. It is creatable if it does not exist, but its parent directory does exist and you can write in that directory.

  In the example below, `foo` is not writable because the parent directory does not exist, even though the user could create such a directory.

```
(file-writable-p "~/no-such-dir/foo")
     ⇒ nil
```

`file-accessible-directory-p` *dirname* [Function]
: This function returns `t` if you have permission to open existing files in the directory whose name as a file is *dirname*; otherwise (e.g., if there is no such directory), it returns `nil`. The value of *dirname* may be either a directory name (such as `/foo/`) or the file name of a file which is a directory (such as `/foo`, without the final slash).

  For example, from the following we deduce that any attempt to read a file in `/foo/` will give an error:

```
(file-accessible-directory-p "/foo")
     ⇒ nil
```

`with-existing-directory` *body...* [Macro]
: This macro ensures that `default-directory` is bound to an existing directory before executing *body*. If `default-directory` already exists, that's preferred, and otherwise some other directory is used. This macro can be useful, for instance, when calling an external command that requires that it's running in a directory that exists. The chosen directory is not guaranteed to be writable.

`access-file` *filename string* [Function]
: If you can read *filename* this function returns `nil`; otherwise it signals an error using *string* as the error message text.

`file-ownership-preserved-p` *filename* **&optional** *group* [Function]
: This function returns `t` if deleting the file *filename* and then creating it anew would keep the file's owner unchanged. It also returns `t` for nonexistent files.

  If the optional argument *group* is non-`nil`, this function also checks that the file's group would be unchanged.

  This function does not follow symbolic links.

`file-modes` *filename* **&optional** *flag* [Function]

This function returns the *mode bits* of *filename*—an integer summarizing its read, write, and execution permissions. This function follows symbolic links. If the file does not exist, the return value is `nil`.

See Section "File permissions" in *The* GNU `Coreutils` *Manual*, for a description of mode bits. For example, if the low-order bit is 1, the file is executable by all users; if the second-lowest-order bit is 1, the file is writable by all users; etc. The highest possible value is 4095 (7777 octal), meaning that everyone has read, write, and execute permission, the SUID bit is set for both others and group, and the sticky bit is set.

By default this function follows symbolic links. However, if the optional argument *flag* is the symbol `nofollow`, this function does not follow *filename* if it is a symbolic link; this can help prevent inadvertently obtaining the mode bits of a file somewhere else, and is more consistent with `file-attributes` (see Section 26.6.4 [File Attributes], page 626).

See Section 26.7 [Changing Files], page 631, for the `set-file-modes` function, which can be used to set these permissions.

```
(file-modes "~/junk/diffs" 'nofollow)
     ⇒ 492                ; Decimal integer.
(format "%o" 492)
     ⇒ "754"              ; Convert to octal.

(set-file-modes "~/junk/diffs" #o666 'nofollow)
     ⇒ nil

$ ls -l diffs
-rw-rw-rw- 1 lewis lewis 3063 Oct 30 16:00 diffs
```

MS-DOS note: On MS-DOS, there is no such thing as an executable file mode bit. So `file-modes` considers a file executable if its name ends in one of the standard executable extensions, such as `.com`, `.bat`, `.exe`, and some others. Files that begin with the POSIX-standard '`#!`' signature, such as shell and Perl scripts, are also considered executable. Directories are also reported as executable, for compatibility with POSIX. These conventions are also followed by `file-attributes` (see Section 26.6.4 [File Attributes], page 626).

26.6.2 Distinguishing Kinds of Files

This section describes how to distinguish various kinds of files, such as directories, symbolic links, and ordinary files.

Symbolic links are ordinarily followed wherever they appear. For example, to interpret the file name `a/b/c`, any of `a`, `a/b`, and `a/b/c` can be symbolic links that are followed, possibly recursively if the link targets are themselves symbolic links. However, a few functions do not follow symbolic links at the end of a file name (`a/b/c` in this example). Such a function is said to *not follow symbolic links*.

`file-symlink-p` *filename* [Function]

If the file *filename* is a symbolic link, this function does not follow it and instead returns its link target as a string. (The link target string is not necessarily the full

absolute file name of the target; determining the full file name that the link points to is nontrivial, see below.)

If the file *filename* is not a symbolic link, or does not exist, or if there is trouble determining whether it is a symbolic link, `file-symlink-p` returns `nil`.

Here are a few examples of using this function:

```
(file-symlink-p "not-a-symlink")
     ⇒ nil
(file-symlink-p "sym-link")
     ⇒ "not-a-symlink"
(file-symlink-p "sym-link2")
     ⇒ "sym-link"
(file-symlink-p "/bin")
     ⇒ "/pub/bin"
```

Note that in the third example, the function returned `sym-link`, but did not proceed to resolve it, although that file is itself a symbolic link. That is because this function does not follow symbolic links—the process of following the symbolic links does not apply to the last component of the file name.

The string that this function returns is what is recorded in the symbolic link; it may or may not include any leading directories. This function does *not* expand the link target to produce a fully-qualified file name, and in particular does not use the leading directories, if any, of the *filename* argument if the link target is not an absolute file name. Here's an example:

```
(file-symlink-p "/foo/bar/baz")
     ⇒ "some-file"
```

Here, although `/foo/bar/baz` was given as a fully-qualified file name, the result is not, and in fact does not have any leading directories at all. And since `some-file` might itself be a symbolic link, you cannot simply prepend leading directories to it, nor even naively use `expand-file-name` (see Section 26.9.4 [File Name Expansion], page 641) to produce its absolute file name.

For this reason, this function is seldom useful if you need to determine more than just the fact that a file is or isn't a symbolic link. If you actually need the file name of the link target, use `file-chase-links` or `file-truename`, described in Section 26.6.3 [Truenames], page 625.

`file-directory-p` *filename* [Function]

This function returns `t` if *filename* is the name of an existing directory. It returns `nil` if *filename* does not name a directory, or if there is trouble determining whether it is a directory. This function follows symbolic links.

```
(file-directory-p "~rms")
     ⇒ t
(file-directory-p "~rms/lewis/files.texi")
     ⇒ nil
(file-directory-p "~rms/lewis/no-such-file")
     ⇒ nil
(file-directory-p "$HOME")
     ⇒ nil
```

```
(file-directory-p
 (substitute-in-file-name "$HOME"))
     ⇒ t
```

`file-regular-p` *filename* [Function]

This function returns `t` if the file *filename* exists and is a regular file (not a directory, named pipe, terminal, or other I/O device). It returns `nil` if *filename* does not exist or is not a regular file, or if there is trouble determining whether it is a regular file. This function follows symbolic links.

26.6.3 Truenames

The *truename* of a file is the name that you get by following symbolic links at all levels until none remain, then simplifying away '`.`' and '`..`' appearing as name components. This results in a sort of canonical name for the file. A file does not always have a unique truename; the number of distinct truenames a file has is equal to the number of hard links to the file. However, truenames are useful because they eliminate symbolic links as a cause of name variation.

`file-truename` *filename* [Function]

This function returns the truename of the file *filename*. If the argument is not an absolute file name, this function first expands it against `default-directory`.

This function does not expand environment variables. Only `substitute-in-file-name` does that. See [Definition of substitute-in-file-name], page 642.

If you may need to follow symbolic links preceding '`..`' appearing as a name component, call `file-truename` without prior direct or indirect calls to `expand-file-name`. Otherwise, the file name component immediately preceding '`..`' will be simplified away before `file-truename` is called. To eliminate the need for a call to `expand-file-name`, `file-truename` handles '`~`' in the same way that `expand-file-name` does.

If the target of a symbolic links has remote file name syntax, `file-truename` returns it quoted. See Section 26.9.4 [Functions that Expand Filenames], page 641.

`file-chase-links` *filename* **&optional** *limit* [Function]

This function follows symbolic links, starting with *filename*, until it finds a file name which is not the name of a symbolic link. Then it returns that file name. This function does *not* follow symbolic links at the level of parent directories.

If you specify a number for *limit*, then after chasing through that many links, the function just returns what it has even if that is still a symbolic link.

To illustrate the difference between `file-chase-links` and `file-truename`, suppose that `/usr/foo` is a symbolic link to the directory `/home/foo`, and `/home/foo/hello` is an ordinary file (or at least, not a symbolic link) or nonexistent. Then we would have:

```
(file-chase-links "/usr/foo/hello")
     ;; This does not follow the links in the parent directories.
     ⇒ "/usr/foo/hello"
(file-truename "/usr/foo/hello")
     ;; Assuming that /home is not a symbolic link.
     ⇒ "/home/foo/hello"
```

`file-equal-p` *file1 file2* [Function]
: This function returns `t` if the files *file1* and *file2* name the same file. This is similar to comparing their truenames, except that remote file names are also handled in an appropriate manner. If *file1* or *file2* does not exist, the return value is unspecified.

`file-name-case-insensitive-p` *filename* [Function]
: Sometimes file names or their parts need to be compared as strings, in which case it's important to know whether the underlying filesystem is case-insensitive. This function returns `t` if file *filename* is on a case-insensitive filesystem. It always returns `t` on MS-DOS and MS-Windows. On Cygwin and macOS, filesystems may or may not be case-insensitive, and the function tries to determine case-sensitivity by a runtime test. If the test is inconclusive, the function returns `t` on Cygwin and `nil` on macOS.

  Currently this function always returns `nil` on platforms other than MS-DOS, MS-Windows, Cygwin, and macOS. It does not detect case-insensitivity of mounted filesystems, such as Samba shares or NFS-mounted Windows volumes. On remote hosts, it assumes `t` for the '`smb`' method. For all other connection methods, runtime tests are performed.

`vc-responsible-backend` *file* [Function]
: This function determines the responsible VC backend of the given *file*. For example, if `emacs.c` is a file tracked by Git, `(vc-responsible-backend "emacs.c")` returns '`Git`'. Note that if *file* is a symbolic link, `vc-responsible-backend` will not resolve it—the backend of the symbolic link file itself is reported. To get the backend VC of the file to which *file* refers, wrap *file* with a symbolic link resolving function such as `file-chase-links`:

```
(vc-responsible-backend (file-chase-links "emacs.c"))
```

26.6.4 File Attributes

This section describes the functions for getting detailed information about a file, including the owner and group numbers, the number of names, the inode number, the size, and the times of access and modification.

`file-newer-than-file-p` *filename1 filename2* [Function]
: This function returns `t` if the file *filename1* is newer than file *filename2*. If *filename1* does not exist, it returns `nil`. If *filename1* does exist, but *filename2* does not, it returns `t`.

  In the following example, assume that the file `aug-19` was written on the 19th, `aug-20` was written on the 20th, and the file `no-file` doesn't exist at all.

```
(file-newer-than-file-p "aug-19" "aug-20")
     ⇒ nil
(file-newer-than-file-p "aug-20" "aug-19")
     ⇒ t
(file-newer-than-file-p "aug-19" "no-file")
     ⇒ t
(file-newer-than-file-p "no-file" "aug-19")
     ⇒ nil
```

`file-has-changed-p` *filename tag* [Function]

This function returns non-`nil` if the time stamp of *filename* has changed since the last call. When called for the first time for some *filename*, it records the last modification time and size of the file, and returns non-`nil` when *filename* exists. Thereafter, when called for the same *filename*, it compares the current time stamp and size with the recorded ones, and returns non-`nil` only if either the time stamp or the size (or both) are different. This is useful when a Lisp program wants to re-read a file whenever it changes. With an optional argument *tag*, which must be a symbol, the size and modification time comparisons are limited to calls with the same tag.

`file-attributes` *filename* **&optional** *id-format* [Function]

This function returns a list of attributes of file *filename*. If the specified file does not exist, it returns `nil`. This function does not follow symbolic links. The optional parameter *id-format* specifies the preferred format of attributes UID and GID (see below)—the valid values are `'string` and `'integer`. The latter is the default, but we plan to change that, so you should specify a non-`nil` value for *id-format* if you use the returned UID or GID.

On GNU platforms when operating on a local file, this function is atomic: if the filesystem is simultaneously being changed by some other process, this function returns the file's attributes either before or after the change. Otherwise this function is not atomic, and might return `nil` if it detects the race condition, or might return a hodgepodge of the previous and current file attributes.

Accessor functions are provided to access the elements in this list. The accessors are mentioned along with the descriptions of the elements below.

The elements of the list, in order, are:

0. `t` for a directory, a string for a symbolic link (the name linked to), or `nil` for a text file (`file-attribute-type`).
1. The number of names the file has (`file-attribute-link-number`). Alternate names, also known as hard links, can be created by using the `add-name-to-file` function (see Section 26.7 [Changing Files], page 631).
2. The file's UID, normally as a string (`file-attribute-user-id`). However, if it does not correspond to a named user, the value is an integer.
3. The file's GID, likewise (`file-attribute-group-id`).
4. The time of last access as a Lisp timestamp (`file-attribute-access-time`). The timestamp is in the style of `current-time` (see Section 42.5 [Time of Day], page 1275) and is truncated to that of the filesystem's timestamp resolution; for example, on some FAT-based filesystems, only the date of last access is recorded, so this time will always hold the midnight of the day of the last access.
5. The time of last modification as a Lisp timestamp (`file-attribute-modification-time`). This is the last time when the file's contents were modified.
6. The time of last status change as a Lisp timestamp (`file-attribute-status-change-time`). This is the time of the last change to the file's access mode bits, its owner and group, and other information recorded in the filesystem for the file, beyond the file's contents.

7. The size of the file in bytes (`file-attribute-size`).
8. The file's modes, as a string of ten letters or dashes, as in '`ls -l`' (`file-attribute-modes`).
9. An unspecified value, present for backward compatibility.
10. The file's inode number (`file-attribute-inode-number`), a nonnegative integer.
11. The filesystem's identifier of the device that the file is on (`file-attribute-device-number`), an integer or a cons cell of two integers. The latter is sometimes used by remote files, in order to distinguish remote filesystems from local ones.

The file's inode and device together give enough information to distinguish any two files on the system—no two files can have the same values for both of these attributes. This tuple that uniquely identifies the file is returned by `file-attribute-file-identifier`.

For example, here are the file attributes for `files.texi`:

```
(file-attributes "files.texi" 'string)
     ⇒  (nil 1 "lh" "users"
         (20614 64019 50040 152000)
         (20000 23 0 0)
         (20614 64555 902289 872000)
         122295 "-rw-rw-rw-"
         t 6473924464520138
         1014478468)
```

and here is how the result is interpreted:

`nil` is neither a directory nor a symbolic link.

`1` has only one name (the name `files.texi` in the current default directory).

`"lh"` is owned by the user with name '`lh`'.

`"users"` is in the group with name '`users`'.

`(20614 64019 50040 152000)`
was last accessed on October 23, 2012, at 20:12:03.050040152 UTC. (This timestamp is `(1351023123050040152 . 1000000000)` if `current-time-list` is `nil`.)

`(20000 23 0 0)`
was last modified on July 15, 2001, at 08:53:43.000000000 UTC. (This timestamp is `(1310720023000000000 . 1000000000)` if `current-time-list` is `nil`.)

`(20614 64555 902289 872000)`
last had its status changed on October 23, 2012, at 20:20:59.902289872 UTC. (This timestamp is `(1351023659902289872 . 1000000000)` if `current-time-list` is `nil`.)

`122295` is 122295 bytes long. (It may not contain 122295 characters, though, if some of the bytes belong to multibyte sequences, and also if the end-of-line format is CR-LF.)

`"-rw-rw-rw-"`
has a mode of read and write access for the owner, group, and world.

`t` is merely a placeholder; it carries no information.

`6473924464520138`
has an inode number of 6473924464520138.

`1014478468`
is on the file-system device whose number is 1014478468.

`file-nlinks` *filename* [Function]
This function returns the number of names (i.e., hard links) that file *filename* has. If the file does not exist, this function returns `nil`. Note that symbolic links have no effect on this function, because they are not considered to be names of the files they link to. This function does not follow symbolic links.

```
$ ls -l foo*
-rw-rw-rw- 2 rms rms 4 Aug 19 01:27 foo
-rw-rw-rw- 2 rms rms 4 Aug 19 01:27 foo1

(file-nlinks "foo")
     ⇒ 2
(file-nlinks "doesnt-exist")
     ⇒ nil
```

26.6.5 Extended File Attributes

On some operating systems, each file can be associated with arbitrary *extended file attributes*. At present, Emacs supports querying and setting two specific sets of extended file attributes: Access Control Lists (ACLs) and SELinux contexts. These extended file attributes are used, on some systems, to impose more sophisticated file access controls than the basic Unix-style permissions discussed in the previous sections.

A detailed explanation of ACLs and SELinux is beyond the scope of this manual. For our purposes, each file can be associated with an *ACL*, which specifies its properties under an ACL-based file control system, and/or an *SELinux context*, which specifies its properties under the SELinux system.

`file-acl` *filename* [Function]
This function returns the ACL for the file *filename*. The exact Lisp representation of the ACL is unspecified (and may change in future Emacs versions), but it is the same as what `set-file-acl` takes for its *acl* argument (see Section 26.7 [Changing Files], page 631).

The underlying ACL implementation is platform-specific; on GNU/Linux and BSD, Emacs uses the POSIX ACL interface, while on MS-Windows Emacs emulates the POSIX ACL interface with native file security APIs.

If ACLs are not supported or the file does not exist, then the return value is `nil`.

`file-selinux-context` *filename* [Function]
: This function returns the SELinux context of the file *filename*, as a list of the form `(user role type range)`. The list elements are the context's user, role, type, and range respectively, as Lisp strings; see the SELinux documentation for details about what these actually mean. The return value has the same form as what `set-file-selinux-context` takes for its *context* argument (see Section 26.7 [Changing Files], page 631).

  If SELinux is not supported or the file does not exist, then the return value is `(nil nil nil nil)`.

`file-extended-attributes` *filename* [Function]
: This function returns an alist of the Emacs-recognized extended attributes of file *filename*. Currently, it serves as a convenient way to retrieve both the ACL and SELinux context; you can then call the function `set-file-extended-attributes`, with the returned alist as its second argument, to apply the same file access attributes to another file (see Section 26.7 [Changing Files], page 631).

  One of the elements is `(acl . acl)`, where *acl* has the same form returned by `file-acl`.

  Another element is `(selinux-context . context)`, where *context* is the SELinux context, in the same form returned by `file-selinux-context`.

26.6.6 Locating Files in Standard Places

This section explains how to search for a file in a list of directories (a *path*), or for an executable file in the standard list of executable file directories.

To search for a user-specific configuration file, See Section 26.9.7 [Standard File Names], page 647, for the `locate-user-emacs-file` function.

`locate-file` *filename path* **&optional** *suffixes predicate* [Function]
: This function searches for a file whose name is *filename* in a list of directories given by *path*, trying the suffixes in *suffixes*. If it finds such a file, it returns the file's absolute file name (see Section 26.9.2 [Relative File Names], page 639); otherwise it returns `nil`.

  The optional argument *suffixes* gives the list of file-name suffixes to append to *filename* when searching. `locate-file` tries each possible directory with each of these suffixes. If *suffixes* is `nil`, or `("")`, then there are no suffixes, and *filename* is used only as-is. Typical values of *suffixes* are `exec-suffixes` (see Section 40.1 [Subprocess Creation], page 1081), `load-suffixes`, `load-file-rep-suffixes` and the return value of the function `get-load-suffixes` (see Section 16.2 [Load Suffixes], page 301).

  Typical values for *path* are `exec-path` (see Section 40.1 [Subprocess Creation], page 1081) when looking for executable programs, or `load-path` (see Section 16.3 [Library Search], page 302) when looking for Lisp files. If *filename* is absolute, *path* has no effect, but the suffixes in *suffixes* are still tried.

  The optional argument *predicate*, if non-`nil`, specifies a predicate function for testing whether a candidate file is suitable. The predicate is passed the candidate file name as its single argument. If *predicate* is `nil` or omitted, `locate-file` uses

`file-readable-p` as the predicate. See Section 26.6.2 [Kinds of Files], page 623, for other useful predicates, e.g., `file-executable-p` and `file-directory-p`.

This function will normally skip directories, so if you want it to find directories, make sure the *predicate* function returns `dir-ok` for them. For example:

```
(locate-file "html" '("/var/www" "/srv") nil
             (lambda (f) (if (file-directory-p f) 'dir-ok)))
```

For compatibility, *predicate* can also be one of the symbols `executable`, `readable`, `writable`, `exists`, or a list of one or more of these symbols.

`executable-find` *program* **&optional** *remote* [Function]

This function searches for the executable file of the named *program* and returns the absolute file name of the executable, including its file-name extensions, if any. It returns `nil` if the file is not found. The function searches in all the directories in `exec-path`, and tries all the file-name extensions in `exec-suffixes` (see Section 40.1 [Subprocess Creation], page 1081).

If *remote* is non-`nil`, and `default-directory` is a remote directory, *program* is searched on the respective remote host.

26.7 Changing File Names and Attributes

The functions in this section rename, copy, delete, link, and set the modes (permissions) of files. Typically, they signal a `file-error` error if they fail to perform their function, reporting the system-dependent error message that describes the reason for the failure. If they fail because a file is missing, they signal a `file-missing` error instead.

For performance, the operating system may cache or alias changes made by these functions instead of writing them immediately to secondary storage. See Section 26.8 [Files and Storage], page 636.

In the functions that have an argument *newname*, if this argument is a directory name it is treated as if the nondirectory part of the source name were appended. Typically, a directory name is one that ends in '`/`' (see Section 26.9.3 [Directory Names], page 640). For example, if the old name is `a/b/c`, the *newname* `d/e/f/` is treated as if it were `d/e/f/c`. This special treatment does not apply if *newname* is not a directory name but names a file that is a directory; for example, the *newname* `d/e/f` is left as-is even if `d/e/f` happens to be a directory.

In the functions that have an argument *newname*, if a file by the name of *newname* already exists, the actions taken depend on the value of the argument *ok-if-already-exists*:

- Signal a `file-already-exists` error if *ok-if-already-exists* is `nil`.
- Request confirmation if *ok-if-already-exists* is a number.
- Replace the old file without confirmation if *ok-if-already-exists* is any other value.

`add-name-to-file` *oldname newname* **&optional** *ok-if-already-exists* [Command]

This function gives the file named *oldname* the additional name *newname*. This means that *newname* becomes a new hard link to *oldname*.

If *newname* is a symbolic link, its directory entry is replaced, not the directory entry it points to. If *oldname* is a symbolic link, this function might or might not follow

the link; it does not follow the link on GNU platforms. If *oldname* is a directory, this function typically fails, although for the superuser on a few old-fashioned non-GNU platforms it can succeed and create a filesystem that is not tree-structured.

In the first part of the following example, we list two files, `foo` and `foo3`.

```
$ ls -li fo*
81908 -rw-rw-rw- 1 rms rms 29 Aug 18 20:32 foo
84302 -rw-rw-rw- 1 rms rms 24 Aug 18 20:31 foo3
```

Now we create a hard link, by calling `add-name-to-file`, then list the files again. This shows two names for one file, `foo` and `foo2`.

```
(add-name-to-file "foo" "foo2")
     ⇒ nil

$ ls -li fo*
81908 -rw-rw-rw- 2 rms rms 29 Aug 18 20:32 foo
81908 -rw-rw-rw- 2 rms rms 29 Aug 18 20:32 foo2
84302 -rw-rw-rw- 1 rms rms 24 Aug 18 20:31 foo3
```

Finally, we evaluate the following:

```
(add-name-to-file "foo" "foo3" t)
```

and list the files again. Now there are three names for one file: `foo`, `foo2`, and `foo3`. The old contents of `foo3` are lost.

```
(add-name-to-file "foo1" "foo3")
     ⇒ nil

$ ls -li fo*
81908 -rw-rw-rw- 3 rms rms 29 Aug 18 20:32 foo
81908 -rw-rw-rw- 3 rms rms 29 Aug 18 20:32 foo2
81908 -rw-rw-rw- 3 rms rms 29 Aug 18 20:32 foo3
```

This function is meaningless on operating systems where multiple names for one file are not allowed. Some systems implement multiple names by copying the file instead.

See also `file-nlinks` in Section 26.6.4 [File Attributes], page 626.

`rename-file` *filename newname* **&optional** *ok-if-already-exists* [Command]

This command renames the file *filename* as *newname*.

If *filename* has additional names aside from *filename*, it continues to have those names. In fact, adding the name *newname* with `add-name-to-file` and then deleting *filename* has the same effect as renaming, aside from momentary intermediate states and treatment of errors, directories and symbolic links.

This command does not follow symbolic links. If *filename* is a symbolic link, this command renames the symbolic link, not the file it points to. If *newname* is a symbolic link, its directory entry is replaced, not the directory entry it points to.

This command does nothing if *filename* and *newname* are the same directory entry, i.e., if they refer to the same parent directory and give the same name within that directory. Otherwise, if *filename* and *newname* name the same file, this command does nothing on POSIX-conforming systems, and removes *filename* on some non-POSIX systems.

If *newname* exists, then it must be an empty directory if *oldname* is a directory and a non-directory otherwise.

`copy-file` *oldname newname* **&optional** *ok-if-already-exists time preserve-uid-gid preserve-permissions* [Command]

This command copies the file *oldname* to *newname*. An error is signaled if *oldname* is not a regular file. If *newname* names a directory, it copies *oldname* into that directory, preserving its final name component.

This function follows symbolic links, except that it does not follow a dangling symbolic link to create *newname*.

If *time* is non-`nil`, then this function gives the new file the same last-modified time that the old one has. (This works on only some operating systems.) If setting the time gets an error, `copy-file` signals a `file-date-error` error. In an interactive call, a prefix argument specifies a non-`nil` value for *time*.

If argument *preserve-uid-gid* is `nil`, we let the operating system decide the user and group ownership of the new file (this is usually set to the user running Emacs). If *preserve-uid-gid* is non-`nil`, we attempt to copy the user and group ownership of the file. This works only on some operating systems, and only if you have the correct permissions to do so.

If the optional argument *preserve-permissions* is non-`nil`, this function copies the file modes (or "permissions") of *oldname* to *newname*, as well as the Access Control List and SELinux context (if any). See Section 26.6 [Information about Files], page 621.

Otherwise, the file modes of *newname* are left unchanged if it is an existing file, and set to those of *oldname*, masked by the default file permissions (see `set-default-file-modes` below), if *newname* is to be newly created. The Access Control List or SELinux context are not copied over in either case.

`make-symbolic-link` *target linkname* **&optional** *ok-if-already-exists* [Command]

This command makes a symbolic link to *target*, named *linkname*. This is like the shell command '`ln -s target linkname`'. The *target* argument is treated only as a string; it need not name an existing file. If *ok-if-already-exists* is an integer, indicating interactive use, then leading '`~`' is expanded and leading '`/:`' is stripped in the *target* string.

If *target* is a relative file name, the resulting symbolic link is interpreted relative to the directory containing the symbolic link. See Section 26.9.2 [Relative File Names], page 639.

If both *target* and *linkname* have remote file name syntax, and if both remote identifications are equal, the symbolic link points to the local file name part of *target*.

This function is not available on systems that don't support symbolic links.

`delete-file` *filename* **&optional** *trash* [Command]

This command deletes the file *filename*. If the file has multiple names, it continues to exist under the other names. If *filename* is a symbolic link, `delete-file` deletes only the symbolic link and not its target.

The command signals a suitable kind of `file-error` error if *filename* cannot be deleted. (On GNU and other POSIX-like systems, a file can be deleted if its directory is writable.) If the file does not exist, this command will not signal any error.

If the optional argument *trash* is non-`nil` and the variable `delete-by-moving-to-trash` is non-`nil`, this command moves the file into the system Trash instead of deleting it. See Section "Miscellaneous File Operations" in *The GNU Emacs Manual*. When called interactively, *trash* is `t` if no prefix argument is given, and `nil` otherwise.

See also `delete-directory` in Section 26.11 [Create/Delete Dirs], page 650.

`set-file-modes` *filename mode* **&optional** *flag* [Command]

This function sets the *file mode* (or *permissions*) of *filename* to *mode*.

By default this function follows symbolic links. However, if the optional argument *flag* is the symbol `nofollow`, this function does not follow *filename* if it is a symbolic link; this can help prevent inadvertently changing the mode bits of a file somewhere else. On platforms that do not support changing mode bits on a symbolic link, this function signals an error when *filename* is a symbolic link and *flag* is `nofollow`.

If called non-interactively, *mode* must be an integer. Only the lowest 12 bits of the integer are used; on most systems, only the lowest 9 bits are meaningful. You can use the Lisp construct for octal numbers to enter *mode*. For example,

```
(set-file-modes "myfile" #o644 'nofollow)
```

specifies that the file should be readable and writable for its owner, readable for group members, and readable for all other users. See Section "File permissions" in *The* GNU `Coreutils` *Manual*, for a description of mode bit specifications.

Interactively, *mode* is read from the minibuffer using `read-file-modes` (see below), which lets the user type in either an integer or a string representing the permissions symbolically.

See Section 26.6.1 [Testing Accessibility], page 621, for the function `file-modes`, which returns the permissions of a file.

`set-default-file-modes` *mode* [Function]

This function sets the default permissions for new files created by Emacs and its subprocesses. Every file created with Emacs initially has these permissions, or a subset of them (`write-region` will not grant execute permissions even if the default file permissions allow execution). On GNU and other POSIX-like systems, the default permissions are given by the bitwise complement of the '`umask`' value, i.e. each bit that is set in the argument *mode* will be *reset* in the default permissions with which Emacs creates files.

The argument *mode* should be an integer which specifies the permissions, similar to `set-file-modes` above. Only the lowest 9 bits are meaningful.

The default file permissions have no effect when you save a modified version of an existing file; saving a file preserves its existing permissions.

`with-file-modes` *mode body...* [Macro]

This macro evaluates the *body* forms with the default permissions for new files temporarily set to *modes* (whose value is as for `set-file-modes` above). When finished, it restores the original default file permissions, and returns the value of the last form in *body*.

This is useful for creating private files, for example.

`default-file-modes` [Function]
This function returns the default file permissions, as an integer.

`read-file-modes` **&optional** *prompt base-file* [Function]
This function reads a set of file mode bits from the minibuffer. The first optional argument *prompt* specifies a non-default prompt. Second second optional argument *base-file* is the name of a file on whose permissions to base the mode bits that this function returns, if what the user types specifies mode bits relative to permissions of an existing file.

If user input represents an octal number, this function returns that number. If it is a complete symbolic specification of mode bits, as in `"u=rwx"`, the function converts it to the equivalent numeric value using `file-modes-symbolic-to-number` and returns the result. If the specification is relative, as in `"o+g"`, then the permissions on which the specification is based are taken from the mode bits of *base-file*. If *base-file* is omitted or `nil`, the function uses `0` as the base mode bits. The complete and relative specifications can be combined, as in `"u+r,g+rx,o+r,g-w"`. See Section "File permissions" in *The* GNU `Coreutils` *Manual*, for a description of file mode specifications.

`file-modes-symbolic-to-number` *modes* **&optional** *base-modes* [Function]
This function converts a symbolic file mode specification in *modes* into the equivalent integer. If the symbolic specification is based on an existing file, that file's mode bits are taken from the optional argument *base-modes*; if that argument is omitted or `nil`, it defaults to 0, i.e., no access rights at all.

`file-modes-number-to-symbolic` *modes* [Function]
This function converts a numeric file mode specification in *modes* into the equivalent string form. The string which this function returns is in the same format produced by the shell command `ls -l` and by `file-attributes`, *not* the symbolic form accepted by `file-modes-symbolic-to-number` and the `chmod` shell command.

`set-file-times` *filename* **&optional** *time flag* [Function]
This function sets the access and modification times of *filename* to *time*. The return value is `t` if the times are successfully set, otherwise it is `nil`. *time* defaults to the current time and must be a time value (see Section 42.5 [Time of Day], page 1275).

By default this function follows symbolic links. However, if the optional argument *flag* is the symbol `nofollow`, this function does not follow *filename* if it is a symbolic link; this can help prevent inadvertently changing the times of a file somewhere else. On platforms that do not support changing times on a symbolic link, this function signals an error when *filename* is a symbolic link and *flag* is `nofollow`.

`set-file-extended-attributes` *filename attribute-alist* [Function]
This function sets the Emacs-recognized extended file attributes for *filename*. The second argument *attribute-alist* should be an alist of the same form returned by `file-extended-attributes`. The return value is `t` if the attributes are successfully set, otherwise it is `nil`. See Section 26.6.5 [Extended Attributes], page 629.

`set-file-selinux-context` *filename context* [Function]
This function sets the SELinux security context for *filename* to *context*. The *context* argument should be a list `(user role type range)`, where each element is a string. See Section 26.6.5 [Extended Attributes], page 629.

The function returns `t` if it succeeds in setting the SELinux context of *filename*. It returns `nil` if the context was not set (e.g., if SELinux is disabled, or if Emacs was compiled without SELinux support).

`set-file-acl` *filename acl* [Function]
This function sets the Access Control List for *filename* to *acl*. The *acl* argument should have the same form returned by the function `file-acl`. See Section 26.6.5 [Extended Attributes], page 629.

The function returns `t` if it successfully sets the ACL of *filename*, `nil` otherwise.

26.8 Files and Secondary Storage

After Emacs changes a file, there are two reasons the changes might not survive later failures of power or media, both having to do with efficiency. First, the operating system might alias written data with data already stored elsewhere on secondary storage until one file or the other is later modified; this will lose both files if the only copy on secondary storage is lost due to media failure. Second, the operating system might not write data to secondary storage immediately, which will lose the data if power is lost.

Although both sorts of failures can largely be avoided by a suitably configured file system, such systems are typically more expensive or less efficient. In more-typical systems, to survive media failure you can copy the file to a different device, and to survive a power failure you can use the `write-region` function with the `write-region-inhibit-fsync` variable set to `nil`. See Section 26.4 [Writing to Files], page 618.

26.9 File Names

Files are generally referred to by their names, in Emacs as elsewhere. File names in Emacs are represented as strings. The functions that operate on a file all expect a file name argument.

In addition to operating on files themselves, Emacs Lisp programs often need to operate on file names; i.e., to take them apart and to use part of a name to construct related file names. This section describes how to manipulate file names.

The functions in this section do not actually access files, so they can operate on file names that do not refer to an existing file or directory.

On MS-DOS and MS-Windows, these functions (like the function that actually operate on files) accept MS-DOS or MS-Windows file-name syntax, where backslashes separate the components, as well as POSIX syntax; but they always return POSIX syntax. This enables Lisp programs to specify file names in POSIX syntax and work properly on all systems without change.[1]

[1] In MS-Windows versions of Emacs compiled for the Cygwin environment, you can use the functions `cygwin-convert-file-name-to-windows` and `cygwin-convert-file-name-from-windows` to convert between the two file-name syntaxes.

26.9.1 File Name Components

The operating system groups files into directories. To specify a file, you must specify the directory and the file's name within that directory. Therefore, Emacs considers a file name as having two main parts: the *directory name* part, and the *nondirectory* part (or *file name within the directory*). Either part may be empty. Concatenating these two parts reproduces the original file name.[2]

On most systems, the directory part is everything up to and including the last slash (backslash is also allowed in input on MS-DOS or MS-Windows); the nondirectory part is the rest.

For some purposes, the nondirectory part is further subdivided into the name proper and the *version number*. On most systems, only backup files have version numbers in their names.

`file-name-directory` *filename* [Function]

This function returns the directory part of *filename*, as a directory name (see Section 26.9.3 [Directory Names], page 640), or `nil` if *filename* does not include a directory part.

On GNU and other POSIX-like systems, a string returned by this function always ends in a slash. On MS-DOS it can also end in a colon.

```
(file-name-directory "lewis/foo")  ; GNU example
     ⇒ "lewis/"
(file-name-directory "foo")        ; GNU example
     ⇒ nil
```

`file-name-nondirectory` *filename* [Function]

This function returns the nondirectory part of *filename*.

```
(file-name-nondirectory "lewis/foo")
     ⇒ "foo"
(file-name-nondirectory "foo")
     ⇒ "foo"
(file-name-nondirectory "lewis/")
     ⇒ ""
```

`file-name-sans-versions` *filename* **&optional** *keep-backup-version* [Function]

This function returns *filename* with any file version numbers, backup version numbers, or trailing tildes discarded.

If *keep-backup-version* is non-`nil`, then true file version numbers understood as such by the file system are discarded from the return value, but backup version numbers are kept.

```
(file-name-sans-versions "~rms/foo.~1~")
     ⇒ "~rms/foo"
(file-name-sans-versions "~rms/foo~")
     ⇒ "~rms/foo"
(file-name-sans-versions "~rms/foo")
     ⇒ "~rms/foo"
```

[2] Emacs follows the GNU convention to use the term *file name* instead of the term *pathname*. We use the term *path* only for search paths, which are lists of directory names.

`file-name-extension` *filename* **&optional** *period* [Function]
This function returns *filename*'s final extension, if any, after applying `file-name-sans-versions` to remove any version/backup part. The extension, in a file name, is the part that follows the last '`.`' in the last name component (minus any version/backup part).

This function returns `nil` for extensionless file names such as `foo`. It returns `""` for null extensions, as in `foo.`. If the last component of a file name begins with a '`.`', that '`.`' doesn't count as the beginning of an extension. Thus, `.emacs`'s extension is `nil`, not '`.emacs`'.

If *period* is non-`nil`, then the returned value includes the period that delimits the extension, and if *filename* has no extension, the value is `""`.

`file-name-with-extension` *filename extension* [Function]
This function returns *filename* with its extension set to *extension*. A single leading dot in the *extension* will be stripped if there is one. For example:

```
(file-name-with-extension "file" "el")
     ⇒ "file.el"
(file-name-with-extension "file" ".el")
     ⇒ "file.el"
(file-name-with-extension "file.c" "el")
     ⇒ "file.el"
```

Note that this function will error if *filename* or *extension* are empty, or if the *filename* is shaped like a directory (i.e., if `directory-name-p` returns non-`nil`).

`file-name-sans-extension` *filename* [Function]
This function returns *filename* minus its extension, if any. The version/backup part, if present, is only removed if the file has an extension. For example,

```
(file-name-sans-extension "foo.lose.c")
     ⇒ "foo.lose"
(file-name-sans-extension "big.hack/foo")
     ⇒ "big.hack/foo"
(file-name-sans-extension "/my/home/.emacs")
     ⇒ "/my/home/.emacs"
(file-name-sans-extension "/my/home/.emacs.el")
     ⇒ "/my/home/.emacs"
(file-name-sans-extension "~/foo.el.~3~")
     ⇒ "~/foo"
(file-name-sans-extension "~/foo.~3~")
     ⇒ "~/foo.~3~"
```

Note that the '`.~3~`' in the two last examples is the backup part, not an extension.

`file-name-base` *filename* [Function]
This function is the composition of `file-name-sans-extension` and `file-name-nondirectory`. For example,

```
(file-name-base "/my/home/foo.c")
    ⇒ "foo"
```

`file-name-split` *filename* [Function]

This function splits a file name into its components, and can be thought of as the inverse of `string-join` with the appropriate directory separator. For example,

```
(file-name-split "/tmp/foo.txt")
    ⇒ ("" "tmp" "foo.txt")
(string-join (file-name-split "/tmp/foo.txt") "/")
    ⇒ "/tmp/foo.txt"
```

26.9.2 Absolute and Relative File Names

All the directories in the file system form a tree starting at the root directory. A file name can specify all the directory names starting from the root of the tree; then it is called an *absolute* file name. Or it can specify the position of the file in the tree relative to a default directory; then it is called a *relative* file name. On GNU and other POSIX-like systems, after any leading '`~`' has been expanded, an absolute file name starts with a '`/`' (see [abbreviate-file-name], page 641), and a relative one does not. On MS-DOS and MS-Windows, an absolute file name starts with a slash or a backslash, or with a drive specification '`x:/`', where *x* is the *drive letter*.

`file-name-absolute-p` *filename* [Function]

This function returns `t` if file *filename* is an absolute file name, `nil` otherwise. A file name is considered to be absolute if its first component is '`~`', or is '`~`*user*' where *user* is a valid login name. In the following examples, assume that there is a user named '`rms`' but no user named '`nosuchuser`'.

```
(file-name-absolute-p "~rms/foo")
     ⇒ t
(file-name-absolute-p "~nosuchuser/foo")
     ⇒ nil
(file-name-absolute-p "rms/foo")
     ⇒ nil
(file-name-absolute-p "/user/rms/foo")
     ⇒ t
```

Given a possibly relative file name, you can expand any leading '`~`' and convert the result to an absolute name using `expand-file-name` (see Section 26.9.4 [File Name Expansion], page 641). This function converts absolute file names to relative names:

`file-relative-name` *filename* **&optional** *directory* [Function]

This function tries to return a relative name that is equivalent to *filename*, assuming the result will be interpreted relative to *directory* (an absolute directory name or directory file name). If *directory* is omitted or `nil`, it defaults to the current buffer's default directory.

On some operating systems, an absolute file name begins with a device name. On such systems, *filename* has no relative equivalent based on *directory* if they start with two different device names. In this case, `file-relative-name` returns *filename* in absolute form.

```
(file-relative-name "/foo/bar" "/foo/")
     ⇒ "bar"
```

```
(file-relative-name "/foo/bar" "/hack/")
     ⇒ "../foo/bar"
```

A file name that is an empty string stands for the current buffer's default directory.

26.9.3 Directory Names

A *directory name* is a string that must name a directory if it names any file at all. A directory is actually a kind of file, and it has a file name (called the *directory file name*), which is related to the directory name but is typically not identical. (This is not quite the same as the usual POSIX terminology.) These two names for the same entity are related by a syntactic transformation. On GNU and other POSIX-like systems, this is simple: to obtain a directory name, append a '`/`' to a directory file name that does not already end in '`/`'. On MS-DOS the relationship is more complicated.

The difference between a directory name and a directory file name is subtle but crucial. When an Emacs variable or function argument is described as being a directory name, a directory file name is not acceptable. When `file-name-directory` returns a string, that is always a directory name.

The following two functions convert between directory names and directory file names. They do nothing special with environment variable substitutions such as '`$HOME`', and the constructs '`~`', '`.`' and '`..`'.

`file-name-as-directory` *filename* [Function]

This function returns a string representing *filename* in a form that the operating system will interpret as the name of a directory (a directory name). On most systems, this means appending a slash to the string (if it does not already end in one).

```
(file-name-as-directory "~rms/lewis")
     ⇒ "~rms/lewis/"
```

`directory-name-p` *filename* [Function]

This function returns non-`nil` if *filename* ends with a directory separator character. This is the forward slash '`/`' on GNU and other POSIX-like systems; MS-Windows and MS-DOS recognize both the forward slash and the backslash '`\`' as directory separators.

`directory-file-name` *dirname* [Function]

This function returns a string representing *dirname* in a form that the operating system will interpret as the name of a file (a directory file name). On most systems, this means removing the final directory separators from the string, unless the string consists entirely of directory separators.

```
(directory-file-name "~lewis/")
     ⇒ "~lewis"
```

`file-name-concat` *directory* **&rest** *components* [Function]

Concatenate *components* to *directory*, inserting a slash before the components if *directory* or the preceding component didn't end with a slash.

```
(file-name-concat "/tmp" "foo")
     ⇒ "/tmp/foo"
```

A *directory* or components that are `nil` or the empty string are ignored—they are filtered out first and do not affect the results in any way.

This is almost the same as using `concat`, but *dirname* (and the non-final components) may or may not end with slash characters, and this function will not double those characters.

To convert a directory name to its abbreviation, use this function:

`abbreviate-file-name` *filename* [Function]

This function returns an abbreviated form of *filename*. It applies the abbreviations specified in `directory-abbrev-alist` (see Section "File Aliases" in *The GNU Emacs Manual*), then substitutes '`~`' for the user's home directory if the argument names a file in the home directory or one of its subdirectories. If the home directory is a root directory, it is not replaced with '`~`', because this does not make the result shorter on many systems.

You can use this function for directory names and for file names, because it recognizes abbreviations even as part of the name.

`file-name-parent-directory` *filename* [Function]

This function returns the directory name of the parent directory of *filename*. If *filename* is at the root directory of the filesystem, it returns `nil`. A relative *filename* is assumed to be relative to `default-directory`, and the return value will also be relative in that case. If the return value is non-`nil`, it ends in a slash.

26.9.4 Functions that Expand Filenames

Expanding a file name means converting a relative file name to an absolute one. Since this is done relative to a default directory, you must specify the default directory as well as the file name to be expanded. It also involves expanding abbreviations like `~/` and eliminating redundancies like `./` and *`name`*`/../`.

`expand-file-name` *filename* **&optional** *directory* [Function]

This function converts *filename* to an absolute file name. If *directory* is supplied, it is the default directory to start with if *filename* is relative and does not start with '`~`'. (The value of *directory* should itself be an absolute directory name or directory file name; it may start with '`~`'.) Otherwise, the current buffer's value of `default-directory` is used. For example:

```
(expand-file-name "foo")
     ⇒ "/xcssun/users/rms/lewis/foo"
(expand-file-name "../foo")
     ⇒ "/xcssun/users/rms/foo"
(expand-file-name "foo" "/usr/spool/")
     ⇒ "/usr/spool/foo"
```

If the part of *filename* before the first slash is '`~`', it expands to your home directory, which is typically specified by the value of the `HOME` environment variable (see Section "General Variables" in *The GNU Emacs Manual*). If the part before the first slash is '`~`*`user`*' and if *user* is a valid login name, it expands to *user*'s home directory. If you do not want this expansion for a relative *filename* that might begin with a literal '`~`',

you can use `(concat (file-name-as-directory directory) filename)` instead of `(expand-file-name filename directory)`.

File names containing '`.`' or '`..`' are simplified to their canonical form:

```
(expand-file-name "bar/../foo")
     ⇒ "/xcssun/users/rms/lewis/foo"
```

In some cases, a leading '`..`' component can remain in the output:

```
(expand-file-name "../home" "/")
     ⇒ "/../home"
```

This is for the sake of filesystems that have the concept of a superroot above the root directory `/`. On other filesystems, `/../` is interpreted exactly the same as `/`.

Expanding `.` or the empty string returns the default directory:

```
(expand-file-name "." "/usr/spool/")
     ⇒ "/usr/spool"
(expand-file-name "" "/usr/spool/")
     ⇒ "/usr/spool"
```

Note that `expand-file-name` does *not* expand environment variables; only `substitute-in-file-name` does that:

```
(expand-file-name "$HOME/foo")
     ⇒ "/xcssun/users/rms/lewis/$HOME/foo"
```

Note also that `expand-file-name` does not follow symbolic links at any level. This results in a difference between the way `file-truename` and `expand-file-name` treat '`..`'. Assuming that '`/tmp/bar`' is a symbolic link to the directory '`/tmp/foo/bar`' we get:

```
(file-truename "/tmp/bar/../myfile")
     ⇒ "/tmp/foo/myfile"
(expand-file-name "/tmp/bar/../myfile")
     ⇒ "/tmp/myfile"
```

If you may need to follow symbolic links preceding '`..`', you should make sure to call `file-truename` without prior direct or indirect calls to `expand-file-name`. See Section 26.6.3 [Truenames], page 625.

`default-directory` [Variable]

The value of this buffer-local variable is the default directory for the current buffer. It should be an absolute directory name; it may start with '`~`'. This variable is buffer-local in every buffer.

`expand-file-name` uses the default directory when its second argument is `nil`.

The value is always a string ending with a slash.

```
default-directory
     ⇒ "/user/lewis/manual/"
```

`substitute-in-file-name` *filename* [Function]

This function replaces environment variable references in *filename* with the environment variable values. Following standard Unix shell syntax, '`$`' is the prefix to

substitute an environment variable value. If the input contains '`$$`', that is converted to '`$`'; this gives the user a way to quote a '`$`'.

The environment variable name is the series of alphanumeric characters (including underscores) that follow the '`$`'. If the character following the '`$`' is a '`{`', then the variable name is everything up to the matching '`}`'.

Calling `substitute-in-file-name` on output produced by `substitute-in-file-name` tends to give incorrect results. For instance, use of '`$$`' to quote a single '`$`' won't work properly, and '`$`' in an environment variable's value could lead to repeated substitution. Therefore, programs that call this function and put the output where it will be passed to this function need to double all '`$`' characters to prevent subsequent incorrect results.

Here we assume that the environment variable `HOME`, which holds the user's home directory, has value '`/xcssun/users/rms`'.

```
(substitute-in-file-name "$HOME/foo")
     ⇒ "/xcssun/users/rms/foo"
```

After substitution, if a '`~`' or a '`/`' appears immediately after another '`/`', the function discards everything before it (up through the immediately preceding '`/`').

```
(substitute-in-file-name "bar/~/foo")
     ⇒ "~/foo"
(substitute-in-file-name "/usr/local/$HOME/foo")
     ⇒ "/xcssun/users/rms/foo"
     ;; /usr/local/ has been discarded.
```

Sometimes, it is not desired to expand file names. In such cases, the file name can be quoted to suppress the expansion, and to handle the file name literally. Quoting happens by prefixing the file name with '`/:`'.

`file-name-quote` *name* [Macro]

This macro adds the quotation prefix '`/:`' to the file *name*. For a local file *name*, it prefixes *name* with '`/:`'. If *name* is a remote file name, the local part of *name* (see Section 26.12 [Magic File Names], page 651) is quoted. If *name* is already a quoted file name, *name* is returned unchanged.

```
(substitute-in-file-name (file-name-quote "bar/~/foo"))
     ⇒ "/:bar/~/foo"

(substitute-in-file-name (file-name-quote "/ssh:host:bar/~/foo"))
     ⇒ "/ssh:host:/:bar/~/foo"
```

The macro cannot be used to suppress file name handlers from magic file names (see Section 26.12 [Magic File Names], page 651).

`file-name-unquote` *name* [Macro]

This macro removes the quotation prefix '`/:`' from the file *name*, if any. If *name* is a remote file name, the local part of *name* is unquoted.

`file-name-quoted-p` *name* [Macro]

This macro returns non-`nil`, when *name* is quoted with the prefix '`/:`'. If *name* is a remote file name, the local part of *name* is checked.

26.9.5 Generating Unique File Names

Some programs need to write temporary files. Here is the usual way to construct a name for such a file:

```
(make-temp-file name-of-application)
```

The job of `make-temp-file` is to prevent two different users or two different jobs from trying to use the exact same file name.

`make-temp-file` *prefix* **&optional** *dir-flag suffix text* [Function]

This function creates a temporary file and returns its name. Emacs creates the temporary file's name by adding to *prefix* some random characters that are different in each Emacs job. The result is guaranteed to be a newly created file, containing *text* if that's given as a string and empty otherwise. On MS-DOS, this function can truncate *prefix* to fit into the 8+3 file-name limits. If *prefix* is a relative file name, it is expanded against `temporary-file-directory`.

```
(make-temp-file "foo")
     ⇒ "/tmp/foo232J6v"
```

When `make-temp-file` returns, the file has been created and is empty. At that point, you should write the intended contents into the file.

If *dir-flag* is non-`nil`, `make-temp-file` creates an empty directory instead of an empty file. It returns the file name, not the directory name, of that directory. See Section 26.9.3 [Directory Names], page 640.

If *suffix* is non-`nil`, `make-temp-file` adds it at the end of the file name.

If *text* is a string, `make-temp-file` inserts it in the file.

To prevent conflicts among different libraries running in the same Emacs, each Lisp program that uses `make-temp-file` should have its own *prefix*. The number added to the end of *prefix* distinguishes between the same application running in different Emacs jobs. Additional added characters permit a large number of distinct names even in one Emacs job.

The default directory for temporary files is controlled by the variable `temporary-file-directory`. This variable gives the user a uniform way to specify the directory for all temporary files. Some programs use `small-temporary-file-directory` instead, if that is non-`nil`. To use it, you should expand the prefix against the proper directory before calling `make-temp-file`.

`temporary-file-directory` [User Option]

This variable specifies the directory name for creating temporary files. Its value should be a directory name (see Section 26.9.3 [Directory Names], page 640), but it is good for Lisp programs to cope if the value is a directory's file name instead. Using the value as the second argument to `expand-file-name` is a good way to achieve that.

The default value is determined in a reasonable way for your operating system; it is based on the `TMPDIR`, `TMP` and `TEMP` environment variables, with a fall-back to a system-dependent name if none of these variables is defined.

Even if you do not use `make-temp-file` to create the temporary file, you should still use this variable to decide which directory to put the file in. However, if you expect

the file to be small, you should use `small-temporary-file-directory` first if that is non-`nil`.

`small-temporary-file-directory` [User Option]
This variable specifies the directory name for creating certain temporary files, which are likely to be small.

If you want to write a temporary file which is likely to be small, you should compute the directory like this:

```
(make-temp-file
  (expand-file-name prefix
                    (or small-temporary-file-directory
                        temporary-file-directory)))
```

`make-temp-name` *base-name* [Function]
This function generates a string that might be a unique file name. The name starts with *base-name*, and has several random characters appended to it, which are different in each Emacs job. It is like `make-temp-file` except that (i) it just constructs a name and does not create a file, (ii) *base-name* should be an absolute file name that is not magic, and (iii) if the returned file name is magic, it might name an existing file. See Section 26.12 [Magic File Names], page 651.

Warning: In most cases, you should not use this function; use `make-temp-file` instead! This function is susceptible to a race condition, between the `make-temp-name` call and the creation of the file, which in some cases may cause a security hole.

Sometimes, it is necessary to create a temporary file on a remote host or a mounted directory. The following two functions support this.

`make-nearby-temp-file` *prefix* **&optional** *dir-flag suffix* [Function]
This function is similar to `make-temp-file`, but it creates a temporary file as close as possible to `default-directory`. If *prefix* is a relative file name, and `default-directory` is a remote file name or located on a mounted file systems, the temporary file is created in the directory returned by the function `temporary-file-directory`. Otherwise, the function `make-temp-file` is used. *prefix*, *dir-flag* and *suffix* have the same meaning as in `make-temp-file`.

```
(let ((default-directory "/ssh:remotehost:"))
  (make-nearby-temp-file "foo"))
     ⇒ "/ssh:remotehost:/tmp/foo232J6v"
```

`temporary-file-directory` [Function]
The directory for writing temporary files via `make-nearby-temp-file`. In case of a remote `default-directory`, this is a directory for temporary files on that remote host. If such a directory does not exist, or `default-directory` ought to be located on a mounted file system (see `mounted-file-systems`), the function returns `default-directory`. For a non-remote and non-mounted `default-directory`, the value of the variable `temporary-file-directory` is returned.

In order to extract the local part of the file's name of a temporary file, use `file-local-name` (see Section 26.12 [Magic File Names], page 651).

26.9.6 File Name Completion

This section describes low-level subroutines for completing a file name. For higher level functions, see Section 21.6.5 [Reading File Names], page 405.

`file-name-all-completions` *partial-filename directory* [Function]

This function returns a list of all possible completions for a file whose name starts with *partial-filename* in directory *directory*. The order of the completions is the order of the files in the directory, which is unpredictable and conveys no useful information.

The argument *partial-filename* must be a file name containing no directory part and no slash (or backslash on some systems). The current buffer's default directory is prepended to *directory*, if *directory* is not absolute.

In the following example, suppose that `~rms/lewis` is the current default directory, and has five files whose names begin with 'f': `foo`, `file~`, `file.c`, `file.c.~1~`, and `file.c.~2~`.

```
(file-name-all-completions "f" "")
     ⇒ ("foo" "file~" "file.c.~2~"
               "file.c.~1~" "file.c")

(file-name-all-completions "fo" "")
     ⇒ ("foo")
```

`file-name-completion` *filename directory* **&optional** *predicate* [Function]

This function completes the file name *filename* in directory *directory*. It returns the longest prefix common to all file names in directory *directory* that start with *filename*. If *predicate* is non-`nil` then it ignores possible completions that don't satisfy *predicate*, after calling that function with one argument, the expanded absolute file name.

If only one match exists and *filename* matches it exactly, the function returns `t`. The function returns `nil` if directory *directory* contains no name starting with *filename*.

In the following example, suppose that the current default directory has five files whose names begin with 'f': `foo`, `file~`, `file.c`, `file.c.~1~`, and `file.c.~2~`.

```
(file-name-completion "fi" "")
     ⇒ "file"

(file-name-completion "file.c.~1" "")
     ⇒ "file.c.~1~"

(file-name-completion "file.c.~1~" "")
     ⇒ t

(file-name-completion "file.c.~3" "")
     ⇒ nil
```

`completion-ignored-extensions` [User Option]

`file-name-completion` usually ignores file names that end in any string in this list. It does not ignore them when all the possible completions end in one of these suffixes. This variable has no effect on `file-name-all-completions`.

A typical value might look like this:

```
completion-ignored-extensions
     ⇒ (".o" ".elc" "~" ".dvi")
```

If an element of `completion-ignored-extensions` ends in a slash '`/`', it signals a directory. The elements which do *not* end in a slash will never match a directory; thus, the above value will not filter out a directory named `foo.elc`.

26.9.7 Standard File Names

Sometimes, an Emacs Lisp program needs to specify a standard file name for a particular use—typically, to hold configuration data specified by the current user. Usually, such files should be located in the directory specified by `user-emacs-directory`, which is typically `~/.config/emacs/` or `~/.emacs.d/` by default (see Section "How Emacs Finds Your Init File" in *The GNU Emacs Manual*). For example, abbrev definitions are stored by default in `~/.config/emacs/abbrev_defs` or `~/.emacs.d/abbrev_defs`. The easiest way to specify such a file name is to use the function `locate-user-emacs-file`.

`locate-user-emacs-file` *base-name* **&optional** *old-name* [Function]

This function returns an absolute file name for an Emacs-specific configuration or data file. The argument `base-name` should be a relative file name. The return value is the absolute name of a file in the directory specified by `user-emacs-directory`; if that directory does not exist, this function creates it.

If the optional argument *old-name* is non-`nil`, it specifies a file in the user's home directory, `~/old-name`. If such a file exists, the return value is the absolute name of that file, instead of the file specified by *base-name*. This argument is intended to be used by Emacs packages to provide backward compatibility. For instance, prior to the introduction of `user-emacs-directory`, the abbrev file was located in `~/.abbrev_defs`. Here is the definition of `abbrev-file-name`:

```
(defcustom abbrev-file-name
  (locate-user-emacs-file "abbrev_defs" ".abbrev_defs")
  "Default name of file from which to read abbrevs."
  ...
  :type 'file)
```

A lower-level function for standardizing file names, which `locate-user-emacs-file` uses as a subroutine, is `convert-standard-filename`.

`convert-standard-filename` *filename* [Function]

This function returns a file name based on *filename*, which fits the conventions of the current operating system.

On GNU and other POSIX-like systems, this simply returns *filename*. On other operating systems, it may enforce system-specific file name conventions; for example, on MS-DOS this function performs a variety of changes to enforce MS-DOS file name limitations, including converting any leading '`.`' to '`_`' and truncating to three characters after the '`.`'.

The recommended way to use this function is to specify a name which fits the conventions of GNU and Unix systems, and pass it to `convert-standard-filename`.

26.10 Contents of Directories

A directory is a kind of file that contains other files entered under various names. Directories are a feature of the file system.

Emacs can list the names of the files in a directory as a Lisp list, or display the names in a buffer using the `ls` shell command. In the latter case, it can optionally display information about each file, depending on the options passed to the `ls` command.

`directory-files` *directory* **&optional** *full-name match-regexp nosort count* [Function]

This function returns a list of the names of the files in the directory *directory*. By default, the list is in alphabetical order.

If *full-name* is non-`nil`, the function returns the files' absolute file names. Otherwise, it returns the names relative to the specified directory.

If *match-regexp* is non-`nil`, this function returns only those file names whose non-directory part contain a match for that regular expression—the other file names are excluded from the list. On case-insensitive filesystems, the regular expression matching is case-insensitive.

If *nosort* is non-`nil`, `directory-files` does not sort the list, so you get the file names in no particular order. Use this if you want the utmost possible speed and don't care what order the files are processed in. If the order of processing is visible to the user, then the user will probably be happier if you do sort the names.

If *count* is non-`nil`, the function will return names of first *count* number of files, or names of all files, whichever occurs first. *count* has to be an integer greater than zero.

```
(directory-files "~lewis")
     ⇒ ("#foo#" "#foo.el#" "." ".."
         "dired-mods.el" "files.texi"
         "files.texi.~1~")
```

An error is signaled if *directory* is not the name of a directory that can be read.

`directory-empty-p` *directory* [Function]

This utility function returns `t` if given *directory* is an accessible directory and it does not contain any files, i.e., is an empty directory. It will ignore '`.`' and '`..`' on systems that return them as files in a directory.

Symbolic links to directories count as directories. See *file-symlink-p* to distinguish symlinks.

`directory-files-recursively` *directory regexp* **&optional** *include-directories predicate follow-symlinks* [Function]

Return all files under *directory* whose names match *regexp*. This function searches the specified *directory* and its sub-directories, recursively, for files whose basenames (i.e., without the leading directories) match the specified *regexp*, and returns a list of the absolute file names of the matching files (see Section 26.9.2 [Relative File Names], page 639). The file names are returned in depth-first order, meaning that files in some sub-directory are returned before the files in its parent directory. In addition, matching files found in each subdirectory are sorted alphabetically by their

basenames. By default, directories whose names match *regexp* are omitted from the list, but if the optional argument *include-directories* is non-`nil`, they are included.

By default, all subdirectories are descended into. If *predicate* is `t`, errors when trying to descend into a subdirectory (for instance, if it's not readable by this user) are ignored. If it's neither `nil` nor `t`, it should be a function that takes one parameter (the subdirectory name) and should return non-`nil` if the directory is to be descended into.

Symbolic links to subdirectories are not followed by default, but if *follow-symlinks* is non-`nil`, they are followed.

`locate-dominating-file` *file name* [Function]

Starting at *file*, go up the directory tree hierarchy looking for the first directory where *name*, a string, exists, and return that directory. If *file* is a file, its directory will serve as the starting point for the search; otherwise *file* should be a directory from which to start. The function looks in the starting directory, then in its parent, then in its parent's parent, etc., until it either finds a directory with *name* or reaches the root directory of the filesystem without finding *name* – in the latter case the function returns `nil`.

The argument `name` can also be a predicate function. The predicate is called for every directory examined by the function, starting from *file* (even if *file* is not a directory). It is called with one argument (the file or directory) and should return non-`nil` if that directory is the one it is looking for.

`file-in-directory-p` *file dir* [Function]

This function returns `t` if *file* is a file in directory *dir*, or in a subdirectory of *dir*. It also returns `t` if *file* and *dir* are the same directory. It compares the truenames of the two directories. If *dir* does not name an existing directory, the return value is `nil`.

`directory-files-and-attributes` *directory* **&optional** *full-name match-regexp nosort id-format count* [Function]

This is similar to `directory-files` in deciding which files to report on and how to report their names. However, instead of returning a list of file names, it returns for each file a list `(filename . attributes)`, where *attributes* is what `file-attributes` returns for that file. The optional argument *id-format* has the same meaning as the corresponding argument to `file-attributes` (see [Definition of file-attributes], page 627).

`directory-files-no-dot-files-regexp` [Constant]

This regular expression matches any file name except '`.`' and '`..`'. More precisely, it matches parts of any nonempty string except those two. It is useful as the *match-regexp* argument to `directory-files` and `directory-files-and-attributes`:

```
(directory-files "/foo" nil directory-files-no-dot-files-regexp)
```

returns `nil`, if directory '`/foo`' is empty.

`file-expand-wildcards` *pattern* **&optional** *full regexp* [Function]

This function expands the wildcard pattern *pattern*, returning a list of file names that match it.

pattern is, by default, a "glob"/wildcard string, e.g., '`"/tmp/*.png"`' or '`"/*/*/foo.png"`', but can also be a regular expression if the optional *regexp* parameter is non-`nil`. In any case, the matches are applied per sub-directory, so a match can't span a parent/sub directory.

If *pattern* is written as an absolute file name, the values are absolute also.

If *pattern* is written as a relative file name, it is interpreted relative to the current default directory. The file names returned are normally also relative to the current default directory. However, if *full* is non-`nil`, they are absolute.

`insert-directory` *file switches* **&optional** *wildcard full-directory-p* [Function]

This function inserts (in the current buffer) a directory listing for directory *file*, formatted with `ls` according to *switches*. It leaves point after the inserted text. *switches* may be a string of options, or a list of strings representing individual options.

The argument *file* may be either a directory or a file specification including wildcard characters. If *wildcard* is non-`nil`, that means treat *file* as a file specification with wildcards.

If *full-directory-p* is non-`nil`, that means the directory listing is expected to show the full contents of a directory. You should specify `t` when *file* is a directory and switches do not contain '`-d`'. (The '`-d`' option to `ls` says to describe a directory itself as a file, rather than showing its contents.)

On most systems, this function works by running a directory listing program whose name is in the variable `insert-directory-program`. If *wildcard* is non-`nil`, it also runs the shell specified by `shell-file-name`, to expand the wildcards.

MS-DOS and MS-Windows systems usually lack the standard Unix program `ls`, so this function emulates the standard Unix program `ls` with Lisp code.

As a technical detail, when *switches* contains the long '`--dired`' option, `insert-directory` treats it specially, for the sake of dired. However, the normally equivalent short '`-D`' option is just passed on to `insert-directory-program`, as any other option.

`insert-directory-program` [Variable]

This variable's value is the program to run to generate a directory listing for the function `insert-directory`. It is ignored on systems which generate the listing with Lisp code.

26.11 Creating, Copying and Deleting Directories

Most Emacs Lisp file-manipulation functions get errors when used on files that are directories. For example, you cannot delete a directory with `delete-file`. These special functions exist to create and delete directories.

`make-directory` *dirname* **&optional** *parents* [Command]

This command creates a directory named *dirname*. If *parents* is non-`nil`, as is always the case in an interactive call, that means to create the parent directories first, if they don't already exist. As a function, `make-directory` returns non-`nil` if *dirname* already exists as a directory and *parents* is non-`nil`, and returns `nil` if it successfully created *dirname*. `mkdir` is an alias for this.

`make-empty-file` *filename* **&optional** *parents* [Command]
This command creates an empty file named *filename*. As `make-directory`, this command creates parent directories if *parents* is non-`nil`. If *filename* already exists, this command signals an error.

`copy-directory` *dirname newname* **&optional** *keep-time parents copy-contents* [Command]
This command copies the directory named *dirname* to *newname*. If *newname* is a directory name, *dirname* will be copied to a subdirectory there. See Section 26.9.3 [Directory Names], page 640.

It always sets the file modes of the copied files to match the corresponding original file.

The third argument *keep-time* non-`nil` means to preserve the modification time of the copied files. A prefix arg makes *keep-time* non-`nil`.

The fourth argument *parents* says whether to create parent directories if they don't exist. Interactively, this happens by default.

The fifth argument *copy-contents*, if non-`nil`, means to copy the contents of *dirname* directly into *newname* if the latter is a directory name, instead of copying *dirname* into it as a subdirectory.

`delete-directory` *dirname* **&optional** *recursive trash* [Command]
This command deletes the directory named *dirname*. The function `delete-file` does not work for files that are directories; you must use `delete-directory` for them. If *recursive* is `nil`, and the directory contains any files, `delete-directory` signals an error. If recursive is non-`nil`, there is no error merely because the directory or its files are deleted by some other process before `delete-directory` gets to them.

`delete-directory` only follows symbolic links at the level of parent directories.

If the optional argument *trash* is non-`nil` and the variable `delete-by-moving-to-trash` is non-`nil`, this command moves the file into the system Trash instead of deleting it. See Section "Miscellaneous File Operations" in *The GNU Emacs Manual*. When called interactively, *trash* is `t` if no prefix argument is given, and `nil` otherwise.

26.12 Making Certain File Names "Magic"

You can implement special handling for certain file names. This is called making those names *magic*. The principal use for this feature is in implementing access to remote files (see Section "Remote Files" in *The GNU Emacs Manual*).

To define a kind of magic file name, you must supply a regular expression to define the class of names (all those that match the regular expression), plus a handler that implements all the primitive Emacs file operations for file names that match.

The variable `file-name-handler-alist` holds a list of handlers, together with regular expressions that determine when to apply each handler. Each element has this form:

```
(regexp . handler)
```

All the Emacs primitives for file access and file name transformation check the given file name against `file-name-handler-alist`. If the file name matches *regexp*, the primitives handle that file by calling *handler*.

The first argument given to *handler* is the name of the primitive, as a symbol; the remaining arguments are the arguments that were passed to that primitive. (The first of these arguments is most often the file name itself.) For example, if you do this:

```
(file-exists-p filename)
```

and *filename* has handler *handler*, then *handler* is called like this:

```
(funcall handler 'file-exists-p filename)
```

When a function takes two or more arguments that must be file names, it checks each of those names for a handler. For example, if you do this:

```
(expand-file-name filename dirname)
```

then it checks for a handler for *filename* and then for a handler for *dirname*. In either case, the *handler* is called like this:

```
(funcall handler 'expand-file-name filename dirname)
```

The *handler* then needs to figure out whether to handle *filename* or *dirname*.

If the specified file name matches more than one handler, the one whose match starts last in the file name gets precedence. This rule is chosen so that handlers for jobs such as uncompression are handled first, before handlers for jobs such as remote file access.

Here are the operations that a magic file name handler gets to handle:

`abbreviate-file-name`, `access-file`,
`add-name-to-file`,
`byte-compiler-base-file-name`,
`copy-directory`, `copy-file`,
`delete-directory`, `delete-file`,
`diff-latest-backup-file`,
`directory-file-name`,
`directory-files`,
`directory-files-and-attributes`,
`dired-compress-file`, `dired-uncache`,
`exec-path`, `expand-file-name`,
`file-accessible-directory-p`,
`file-acl`,
`file-attributes`,
`file-directory-p`,
`file-equal-p`,
`file-executable-p`, `file-exists-p`,
`file-in-directory-p`,
`file-local-copy`, `file-locked-p`,
`file-modes`, `file-name-all-completions`,
`file-name-as-directory`,
`file-name-case-insensitive-p`,
`file-name-completion`,
`file-name-directory`,
`file-name-nondirectory`,
`file-name-sans-versions`, `file-newer-than-file-p`,
`file-notify-add-watch`, `file-notify-rm-watch`,

`file-notify-valid-p`,
`file-ownership-preserved-p`,
`file-readable-p`, `file-regular-p`,
`file-remote-p`, `file-selinux-context`,
`file-symlink-p`, `file-system-info`,
`file-truename`, `file-writable-p`,
`find-backup-file-name`,
`get-file-buffer`,
`insert-directory`,
`insert-file-contents`,
`list-system-processes`,
`load`, `lock-file`,
`make-auto-save-file-name`,
`make-directory`,
`make-lock-file-name`,
`make-nearby-temp-file`,
`make-process`,
`make-symbolic-link`,
`memory-info`, `process-attributes`, `process-file`,
`rename-file`, `set-file-acl`, `set-file-modes`,
`set-file-selinux-context`, `set-file-times`,
`set-visited-file-modtime`, `shell-command`,
`start-file-process`,
`substitute-in-file-name`,
`temporary-file-directory`,
`unhandled-file-name-directory`,
`unlock-file`,
`vc-registered`,
`verify-visited-file-modtime`,
`write-region`.

Handlers for `insert-file-contents` typically need to clear the buffer's modified flag, with `(set-buffer-modified-p nil)`, if the *visit* argument is non-`nil`. This also has the effect of unlocking the buffer if it is locked.

The handler function must handle all of the above operations, and possibly others to be added in the future. It need not implement all these operations itself—when it has nothing special to do for a certain operation, it can reinvoke the primitive, to handle the operation in the usual way. It should always reinvoke the primitive for an operation it does not recognize. Here's one way to do this:

```
(defun my-file-handler (operation &rest args)
  ;; First check for the specific operations
  ;; that we have special handling for.
  (cond ((eq operation 'insert-file-contents) ...)
        ((eq operation 'write-region) ...)
        ...
        ;; Handle any operation we don't know about.
        (t (let ((inhibit-file-name-handlers
                  (cons 'my-file-handler
                        (and (eq inhibit-file-name-operation operation)
```

```
                                inhibit-file-name-handlers)))
                    (inhibit-file-name-operation operation))
                (apply operation args)))))
```

When a handler function decides to call the ordinary Emacs primitive for the operation at hand, it needs to prevent the primitive from calling the same handler once again, thus leading to an infinite recursion. The example above shows how to do this, with the variables `inhibit-file-name-handlers` and `inhibit-file-name-operation`. Be careful to use them exactly as shown above; the details are crucial for proper behavior in the case of multiple handlers, and for operations that have two file names that may each have handlers.

Handlers that don't really do anything special for actual access to the file—such as the ones that implement completion of host names for remote file names—should have a non-`nil` `safe-magic` property. For instance, Emacs normally protects directory names it finds in `PATH` from becoming magic, if they look like magic file names, by prefixing them with '`/:`'. But if the handler that would be used for them has a non-`nil` `safe-magic` property, the '`/:`' is not added.

A file name handler can have an `operations` property to declare which operations it handles in a nontrivial way. If this property has a non-`nil` value, it should be a list of operations; then only those operations will call the handler. This avoids inefficiency, but its main purpose is for autoloaded handler functions, so that they won't be loaded except when they have real work to do.

Simply deferring all operations to the usual primitives does not work. For instance, if the file name handler applies to `file-exists-p`, then it must handle `load` itself, because the usual `load` code won't work properly in that case. However, if the handler uses the `operations` property to say it doesn't handle `file-exists-p`, then it need not handle `load` nontrivially.

`inhibit-file-name-handlers` [Variable]

This variable holds a list of handlers whose use is presently inhibited for a certain operation.

`inhibit-file-name-operation` [Variable]

The operation for which certain handlers are presently inhibited.

`find-file-name-handler` *file operation* [Function]

This function returns the handler function for file name *file*, or `nil` if there is none. The argument *operation* should be the operation to be performed on the file—the value you will pass to the handler as its first argument when you call it. If *operation* equals `inhibit-file-name-operation`, or if it is not found in the `operations` property of the handler, this function returns `nil`.

`file-local-copy` *filename* [Function]

This function copies file *filename* to an ordinary non-magic file on the local machine, if it isn't on the local machine already. Magic file names should handle the `file-local-copy` operation if they refer to files on other machines. A magic file name that is used for other purposes than remote file access should not handle `file-local-copy`; then this function will treat the file as local.

If *filename* is local, whether magic or not, this function does nothing and returns `nil`. Otherwise it returns the file name of the local copy file.

`file-remote-p` *filename* **&optional** *identification connected* [Function]
This function tests whether *filename* is a remote file. If *filename* is local (not remote), the return value is `nil`. If *filename* is indeed remote, the return value is a string that identifies the remote system.

This identifier string can include a host name and a user name, as well as characters designating the method used to access the remote system. For example, the remote identifier string for the file name `/sudo::/some/file` is `/sudo:root@localhost:`.

If `file-remote-p` returns the same identifier for two different file names, that means they are stored on the same file system and can be accessed locally with respect to each other. This means, for example, that it is possible to start a remote process accessing both files at the same time. Implementers of file name handlers need to ensure this principle is valid.

identification specifies which part of the identifier shall be returned as string. *identification* can be the symbol `method`, `user` or `host`; any other value is handled like `nil` and means to return the complete identifier string. In the example above, the remote `user` identifier string would be `root`.

If *connected* is non-`nil`, this function returns `nil` even if *filename* is remote, if Emacs has no network connection to its host. This is useful when you want to avoid the delay of making connections when they don't exist.

`unhandled-file-name-directory` *filename* [Function]
This function returns the name of a directory that is not magic. For a non-magic *filename* it returns the corresponding directory name (see Section 26.9.3 [Directory Names], page 640). For a magic *filename*, it invokes the file name handler, which therefore decides what value to return. If *filename* is not accessible from a local process, then the file name handler should indicate that by returning `nil`.

This is useful for running a subprocess; every subprocess must have a non-magic directory to serve as its current directory, and this function is a good way to come up with one.

`file-local-name` *filename* [Function]
This function returns the *local part* of *filename*. This is the part of the file's name that identifies it on the remote host, and is typically obtained by removing from the remote file name the parts that specify the remote host and the method of accessing it. For example:

```
(file-local-name "/ssh:user@host:/foo/bar")
     ⇒ "/foo/bar"
```

For a remote *filename*, this function returns a file name which could be used directly as an argument of a remote process (see Section 40.4 [Asynchronous Processes], page 1089, and see Section 40.3 [Synchronous Processes], page 1084), and as the program to run on the remote host. If *filename* is local, this function returns it unchanged.

`remote-file-name-inhibit-cache` [User Option]
The attributes of remote files can be cached for better performance. If they are changed outside of Emacs's control, the cached values become invalid, and must be reread.

When this variable is set to `nil`, cached values are never expired. Use this setting with caution, only if you are sure nothing other than Emacs ever changes the remote files. If it is set to `t`, cached values are never used. This is the safest value, but could result in performance degradation.

A compromise is to set it to a positive number. This means that cached values are used for that amount of seconds since they were cached. If a remote file is checked regularly, it might be a good idea to let-bind this variable to a value less than the time period between consecutive checks. For example:

```
(defun display-time-file-nonempty-p (file)
  (let ((remote-file-name-inhibit-cache
         (- display-time-interval 5)))
    (and (file-exists-p file)
         (< 0 (file-attribute-size
               (file-attributes
                (file-chase-links file)))))))
```

26.13 File Format Conversion

Emacs performs several steps to convert the data in a buffer (text, text properties, and possibly other information) to and from a representation suitable for storing into a file. This section describes the fundamental functions that perform this *format conversion*, namely `insert-file-contents` for reading a file into a buffer, and `write-region` for writing a buffer into a file.

26.13.1 Overview

The function `insert-file-contents`:

- initially, inserts bytes from the file into the buffer;
- decodes bytes to characters as appropriate;
- processes formats as defined by entries in `format-alist`; and
- calls functions in `after-insert-file-functions`.

The function `write-region`:

- initially, calls functions in `write-region-annotate-functions`;
- processes formats as defined by entries in `format-alist`;
- encodes characters to bytes as appropriate; and
- modifies the file with the bytes.

This shows the symmetry of the lowest-level operations; reading and writing handle things in opposite order. The rest of this section describes the two facilities surrounding the three variables named above, as well as some related functions. Section 34.10 [Coding Systems], page 980, for details on character encoding and decoding.

26.13.2 Round-Trip Specification

The most general of the two facilities is controlled by the variable `format-alist`, a list of *file format* specifications, which describe textual representations used in files for the data in an Emacs buffer. The descriptions for reading and writing are paired, which is why

we call this "round-trip" specification (see Section 26.13.3 [Format Conversion Piecemeal], page 659, for non-paired specification).

`format-alist` [Variable]

This list contains one format definition for each defined file format. Each format definition is a list of this form:

```
(name doc-string regexp from-fn to-fn modify mode-fn preserve)
```

Here is what the elements in a format definition mean:

name The name of this format.

doc-string A documentation string for the format.

regexp A regular expression which is used to recognize files represented in this format. If `nil`, the format is never applied automatically.

from-fn A shell command or function to decode data in this format (to convert file data into the usual Emacs data representation).

A shell command is represented as a string; Emacs runs the command as a filter to perform the conversion.

If *from-fn* is a function, it is called with two arguments, *begin* and *end*, which specify the part of the buffer it should convert. It should convert the text by editing it in place. Since this can change the length of the text, *from-fn* should return the modified end position.

One responsibility of *from-fn* is to make sure that the beginning of the file no longer matches *regexp*. Otherwise it is likely to get called again. Also, *from-fn* must not involve buffers or files other than the one being decoded, otherwise the internal buffer used for formatting might be overwritten.

to-fn A shell command or function to encode data in this format—that is, to convert the usual Emacs data representation into this format.

If *to-fn* is a string, it is a shell command; Emacs runs the command as a filter to perform the conversion.

If *to-fn* is a function, it is called with three arguments: *begin* and *end*, which specify the part of the buffer it should convert, and *buffer*, which specifies which buffer. There are two ways it can do the conversion:

- By editing the buffer in place. In this case, *to-fn* should return the end-position of the range of text, as modified.
- By returning a list of annotations. This is a list of elements of the form `(position . string)`, where *position* is an integer specifying the relative position in the text to be written, and *string* is the annotation to add there. The list must be sorted in order of position when *to-fn* returns it.

  When `write-region` actually writes the text from the buffer to the file, it intermixes the specified annotations at the corresponding positions. All this takes place without modifying the buffer.

to-fn must not involve buffers or files other than the one being encoded, otherwise the internal buffer used for formatting might be overwritten.

modify A flag, `t` if the encoding function modifies the buffer, and `nil` if it works by returning a list of annotations.

mode-fn A minor-mode function to call after visiting a file converted from this format. The function is called with one argument, the integer 1; that tells a minor-mode function to enable the mode.

preserve A flag, `t` if `format-write-file` should not remove this format from `buffer-file-format`.

The function `insert-file-contents` automatically recognizes file formats when it reads the specified file. It checks the text of the beginning of the file against the regular expressions of the format definitions, and if it finds a match, it calls the decoding function for that format. Then it checks all the known formats over again. It keeps checking them until none of them is applicable.

Visiting a file, with `find-file-noselect` or the commands that use it, performs conversion likewise (because it calls `insert-file-contents`); it also calls the mode function for each format that it decodes. It stores a list of the format names in the buffer-local variable `buffer-file-format`.

`buffer-file-format` [Variable]
This variable states the format of the visited file. More precisely, this is a list of the file format names that were decoded in the course of visiting the current buffer's file. It is always buffer-local in all buffers.

When `write-region` writes data into a file, it first calls the encoding functions for the formats listed in `buffer-file-format`, in the order of appearance in the list.

`format-write-file` *file format* **&optional** *confirm* [Command]
This command writes the current buffer contents into the file *file* in a format based on *format*, which is a list of format names. It constructs the actual format starting from *format*, then appending any elements from the value of `buffer-file-format` with a non-`nil` *preserve* flag (see above), if they are not already present in *format*. It then updates `buffer-file-format` with this format, making it the default for future saves. Except for the *format* argument, this command is similar to `write-file`. In particular, *confirm* has the same meaning and interactive treatment as the corresponding argument to `write-file`. See [Definition of write-file], page 614.

`format-find-file` *file format* [Command]
This command finds the file *file*, converting it according to format *format*. It also makes *format* the default if the buffer is saved later.

The argument *format* is a list of format names. If *format* is `nil`, no conversion takes place. Interactively, typing just `RET` for *format* specifies `nil`.

`format-insert-file` *file format* **&optional** *beg end* [Command]
This command inserts the contents of file *file*, converting it according to format *format*. If *beg* and *end* are non-`nil`, they specify which part of the file to read, as in `insert-file-contents` (see Section 26.3 [Reading from Files], page 616).

The return value is like what `insert-file-contents` returns: a list of the absolute file name and the length of the data inserted (after conversion).

The argument *format* is a list of format names. If *format* is `nil`, no conversion takes place. Interactively, typing just `RET` for *format* specifies `nil`.

`buffer-auto-save-file-format` [Variable]

This variable specifies the format to use for auto-saving. Its value is a list of format names, just like the value of `buffer-file-format`; however, it is used instead of `buffer-file-format` for writing auto-save files. If the value is `t`, the default, auto-saving uses the same format as a regular save in the same buffer. This variable is always buffer-local in all buffers.

26.13.3 Piecemeal Specification

In contrast to the round-trip specification described in the previous subsection (see Section 26.13.2 [Format Conversion Round-Trip], page 656), you can use the variables `after-insert-file-functions` and `write-region-annotate-functions` to separately control the respective reading and writing conversions.

Conversion starts with one representation and produces another representation. When there is only one conversion to do, there is no conflict about what to start with. However, when there are multiple conversions involved, conflict may arise when two conversions need to start with the same data.

This situation is best understood in the context of converting text properties during `write-region`. For example, the character at position 42 in a buffer is '`X`' with a text property `foo`. If the conversion for `foo` is done by inserting into the buffer, say, '`FOO:`', then that changes the character at position 42 from '`X`' to '`F`'. The next conversion will start with the wrong data straight away.

To avoid conflict, cooperative conversions do not modify the buffer, but instead specify *annotations*, a list of elements of the form `(`*`position`*` . `*`string`*`)`, sorted in order of increasing *position*.

If there is more than one conversion, `write-region` merges their annotations destructively into one sorted list. Later, when the text from the buffer is actually written to the file, it intermixes the specified annotations at the corresponding positions. All this takes place without modifying the buffer.

In contrast, when reading, the annotations intermixed with the text are handled immediately. `insert-file-contents` sets point to the beginning of some text to be converted, then calls the conversion functions with the length of that text. These functions should always return with point at the beginning of the inserted text. This approach makes sense for reading because annotations removed by the first converter can't be mistakenly processed by a later converter. Each conversion function should scan for the annotations it recognizes, remove the annotation, modify the buffer text (to set a text property, for example), and return the updated length of the text, as it stands after those changes. The value returned by one function becomes the argument to the next function.

`write-region-annotate-functions` [Variable]

A list of functions for `write-region` to call. Each function in the list is called with two arguments: the start and end of the region to be written. These functions should not alter the contents of the buffer. Instead, they should return annotations.

As a special case, a function may return with a different buffer current. Emacs takes this to mean that the current buffer contains altered text to be output. It therefore

changes the *start* and *end* arguments of the `write-region` call, giving them the values of `point-min` and `point-max` in the new buffer, respectively. It also discards all previous annotations, because they should have been dealt with by this function.

`write-region-post-annotation-function` [Variable]

The value of this variable, if non-`nil`, should be a function. This function is called, with no arguments, after `write-region` has completed.

If any function in `write-region-annotate-functions` returns with a different buffer current, Emacs calls `write-region-post-annotation-function` more than once. Emacs calls it with the last buffer that was current, and again with the buffer before that, and so on back to the original buffer.

Thus, a function in `write-region-annotate-functions` can create a buffer, give this variable the local value of `kill-buffer` in that buffer, set up the buffer with altered text, and make the buffer current. The buffer will be killed after `write-region` is done.

`after-insert-file-functions` [Variable]

Each function in this list is called by `insert-file-contents` with one argument, the number of characters inserted, and with point at the beginning of the inserted text. Each function should leave point unchanged, and return the new character count describing the inserted text as modified by the function.

We invite users to write Lisp programs to store and retrieve text properties in files, using these hooks, and thus to experiment with various data formats and find good ones. Eventually we hope users will produce good, general extensions we can install in Emacs.

We suggest not trying to handle arbitrary Lisp objects as text property names or values—because a program that general is probably difficult to write, and slow. Instead, choose a set of possible data types that are reasonably flexible, and not too hard to encode.

27 Backups and Auto-Saving

Backup files and auto-save files are two methods by which Emacs tries to protect the user from the consequences of crashes or of the user's own errors. Auto-saving preserves the text from earlier in the current editing session; backup files preserve file contents prior to the current session.

27.1 Backup Files

A *backup file* is a copy of the old contents of a file you are editing. Emacs makes a backup file the first time you save a buffer into its visited file. Thus, normally, the backup file contains the contents of the file as it was before the current editing session. The contents of the backup file normally remain unchanged once it exists.

Backups are usually made by renaming the visited file to a new name. Optionally, you can specify that backup files should be made by copying the visited file. This choice makes a difference for files with multiple names; it also can affect whether the edited file remains owned by the original owner or becomes owned by the user editing it.

By default, Emacs makes a single backup file for each file edited. You can alternatively request numbered backups; then each new backup file gets a new name. You can delete old numbered backups when you don't want them any more, or Emacs can delete them automatically.

For performance, the operating system may not write the backup file's contents to secondary storage immediately, or may alias the backup data with the original until one or the other is later modified. See Section 26.8 [Files and Storage], page 636.

27.1.1 Making Backup Files

`backup-buffer` [Function]

This function makes a backup of the file visited by the current buffer, if appropriate. It is called by `save-buffer` before saving the buffer the first time.

If a backup was made by renaming, the return value is a cons cell of the form (*modes extra-alist backupname*), where *modes* are the mode bits of the original file, as returned by `file-modes` (see Section 26.6.1 [Testing Accessibility], page 621), *extra-alist* is an alist describing the original file's extended attributes, as returned by `file-extended-attributes` (see Section 26.6.5 [Extended Attributes], page 629), and *backupname* is the name of the backup.

In all other cases (i.e., if a backup was made by copying or if no backup was made), this function returns `nil`.

`buffer-backed-up` [Variable]

This buffer-local variable says whether this buffer's file has been backed up on account of this buffer. If it is non-`nil`, the backup file has been written. Otherwise, the file should be backed up when it is next saved (if backups are enabled). This is a permanent local; `kill-all-local-variables` does not alter it.

`make-backup-files` [User Option]

This variable determines whether or not to make backup files. If it is non-`nil`, then Emacs creates a backup of each file when it is saved for the first time—provided that `backup-inhibited` is `nil` (see below).

The following example shows how to change the `make-backup-files` variable only in the Rmail buffers and not elsewhere. Setting it `nil` stops Emacs from making backups of these files, which may save disk space. (You would put this code in your init file.)

```
(add-hook 'rmail-mode-hook
          (lambda () (setq-local make-backup-files nil)))
```

`backup-enable-predicate` [Variable]

This variable's value is a function to be called on certain occasions to decide whether a file should have backup files. The function receives one argument, an absolute file name to consider. If the function returns `nil`, backups are disabled for that file. Otherwise, the other variables in this section say whether and how to make backups.

The default value is `normal-backup-enable-predicate`, which checks for files in `temporary-file-directory` and `small-temporary-file-directory`.

`backup-inhibited` [Variable]

If this variable is non-`nil`, backups are inhibited. It records the result of testing `backup-enable-predicate` on the visited file name. It can also coherently be used by other mechanisms that inhibit backups based on which file is visited. For example, VC sets this variable non-`nil` to prevent making backups for files managed with a version control system.

This is a permanent local, so that changing the major mode does not lose its value. Major modes should not set this variable—they should set `make-backup-files` instead.

`backup-directory-alist` [User Option]

This variable's value is an alist of filename patterns and backup directories. Each element looks like

```
(regexp . directory)
```

Backups of files with names matching *regexp* will be made in *directory*. *directory* may be relative or absolute. If it is absolute, so that all matching files are backed up into the same directory, the file names in this directory will be the full name of the file backed up with all directory separators changed to '`!`' to prevent clashes. This will not work correctly if your filesystem truncates the resulting name.

For the common case of all backups going into one directory, the alist should contain a single element pairing '`"."`' with the appropriate directory.

If this variable is `nil` (the default), or it fails to match a filename, the backup is made in the original file's directory.

On MS-DOS filesystems without long names this variable is always ignored.

`make-backup-file-name-function` [User Option]

This variable's value is a function to use for making backup file names. The function `make-backup-file-name` calls it. See Section 27.1.4 [Naming Backup Files], page 665.

This could be buffer-local to do something special for specific files. If you change it, you may need to change `backup-file-name-p` and `file-name-sans-versions` too.

27.1.2 Backup by Renaming or by Copying?

There are two ways that Emacs can make a backup file:

- Emacs can rename the original file so that it becomes a backup file, and then write the buffer being saved into a new file. After this procedure, any other names (i.e., hard links) of the original file now refer to the backup file. The new file is owned by the user doing the editing, and its group is the default for new files written by the user in that directory.
- Emacs can copy the original file into a backup file, and then overwrite the original file with new contents. After this procedure, any other names (i.e., hard links) of the original file continue to refer to the current (updated) version of the file. The file's owner and group will be unchanged.

The first method, renaming, is the default.

The variable `backup-by-copying`, if non-`nil`, says to use the second method, which is to copy the original file and overwrite it with the new buffer contents. The variable `file-precious-flag`, if non-`nil`, also has this effect (as a sideline of its main significance). See Section 26.2 [Saving Buffers], page 614.

`backup-by-copying` [User Option]
: If this variable is non-`nil`, Emacs always makes backup files by copying. The default is `nil`.

The following three variables, when non-`nil`, cause the second method to be used in certain special cases. They have no effect on the treatment of files that don't fall into the special cases.

`backup-by-copying-when-linked` [User Option]
: If this variable is non-`nil`, Emacs makes backups by copying for files with multiple names (hard links). The default is `nil`.

  This variable is significant only if `backup-by-copying` is `nil`, since copying is always used when that variable is non-`nil`.

`backup-by-copying-when-mismatch` [User Option]
: If this variable is non-`nil` (the default), Emacs makes backups by copying in cases where renaming would change either the owner or the group of the file.

  The value has no effect when renaming would not alter the owner or group of the file; that is, for files which are owned by the user and whose group matches the default for a new file created there by the user.

  This variable is significant only if `backup-by-copying` is `nil`, since copying is always used when that variable is non-`nil`.

`backup-by-copying-when-privileged-mismatch` [User Option]
: This variable, if non-`nil`, specifies the same behavior as `backup-by-copying-when-mismatch`, but only for certain user-id and group-id values: namely, those less than or equal to a certain number. You set this variable to that number.

Thus, if you set `backup-by-copying-when-privileged-mismatch` to 0, backup by copying is done for the superuser and group 0 only, when necessary to prevent a change in the owner of the file.

The default is 200.

27.1.3 Making and Deleting Numbered Backup Files

If a file's name is `foo`, the names of its numbered backup versions are `foo.~`*v*`~`, for various integers *v*, like this: `foo.~1~`, `foo.~2~`, `foo.~3~`, ..., `foo.~259~`, and so on.

`version-control` [User Option]

This variable controls whether to make a single non-numbered backup file or multiple numbered backups.

`nil` Make numbered backups if the visited file already has numbered backups; otherwise, do not. This is the default.

`never` Do not make numbered backups.

anything else
Make numbered backups.

The use of numbered backups ultimately leads to a large number of backup versions, which must then be deleted. Emacs can do this automatically or it can ask the user whether to delete them.

`kept-new-versions` [User Option]

The value of this variable is the number of newest versions to keep when a new numbered backup is made. The newly made backup is included in the count. The default value is 2.

`kept-old-versions` [User Option]

The value of this variable is the number of oldest versions to keep when a new numbered backup is made. The default value is 2.

If there are backups numbered 1, 2, 3, 5, and 7, and both of these variables have the value 2, then the backups numbered 1 and 2 are kept as old versions and those numbered 5 and 7 are kept as new versions; backup version 3 is excess. The function `find-backup-file-name` (see Section 27.1.4 [Backup Names], page 665) is responsible for determining which backup versions to delete, but does not delete them itself.

`delete-old-versions` [User Option]

If this variable is `t`, then saving a file deletes excess backup versions silently. If it is `nil`, that means to ask for confirmation before deleting excess backups. Otherwise, they are not deleted at all.

`dired-kept-versions` [User Option]

This variable specifies how many of the newest backup versions to keep in the Dired command `.` (`dired-clean-directory`). That's the same thing `kept-new-versions` specifies when you make a new backup file. The default is 2.

27.1.4 Naming Backup Files

The functions in this section are documented mainly because you can customize the naming conventions for backup files by redefining them. If you change one, you probably need to change the rest.

`backup-file-name-p` *filename* [Function]

This function returns a non-`nil` value if *filename* is a possible name for a backup file. It just checks the name, not whether a file with the name *filename* exists.

```
(backup-file-name-p "foo")
     ⇒ nil
(backup-file-name-p "foo~")
     ⇒ 3
```

The standard definition of this function is as follows:

```
(defun backup-file-name-p (file)
  "Return non-nil if FILE is a backup file \
name (numeric or not)..."
  (string-match "~\\'" file))
```

Thus, the function returns a non-`nil` value if the file name ends with a '`~`'. (We use a backslash to split the documentation string's first line into two lines in the text, but produce just one line in the string itself.)

This simple expression is placed in a separate function to make it easy to redefine for customization.

`make-backup-file-name` *filename* [Function]

This function returns a string that is the name to use for a non-numbered backup file for file *filename*. On Unix, this is just *filename* with a tilde appended.

The standard definition of this function, on most operating systems, is as follows:

```
(defun make-backup-file-name (file)
  "Create the non-numeric backup file name for FILE..."
  (concat file "~"))
```

You can change the backup-file naming convention by redefining this function. The following example redefines `make-backup-file-name` to prepend a '`.`' in addition to appending a tilde:

```
(defun make-backup-file-name (filename)
  (expand-file-name
    (concat "." (file-name-nondirectory filename) "~")
    (file-name-directory filename)))

(make-backup-file-name "backups.texi")
     ⇒ ".backups.texi~"
```

Some parts of Emacs, including some Dired commands, assume that backup file names end with '`~`'. If you do not follow that convention, it will not cause serious problems, but these commands may give less-than-desirable results.

`find-backup-file-name` *filename* [Function]

This function computes the file name for a new backup file for *filename*. It may also propose certain existing backup files for deletion. `find-backup-file-name` returns a list whose CAR is the name for the new backup file and whose CDR is a list of backup

files whose deletion is proposed. The value can also be `nil`, which means not to make a backup.

Two variables, `kept-old-versions` and `kept-new-versions`, determine which backup versions should be kept. This function keeps those versions by excluding them from the CDR of the value. See Section 27.1.3 [Numbered Backups], page 664.

In this example, the value says that `~rms/foo.~5~` is the name to use for the new backup file, and `~rms/foo.~3~` is an excess version that the caller should consider deleting now.

```
(find-backup-file-name "~rms/foo")
     ⇒ ("~rms/foo.~5~" "~rms/foo.~3~")
```

`file-backup-file-names` *filename* [Function]

This function returns a list of all the backup file names for *filename*, or `nil` if there are none. The files are sorted by modification time, descending, so that the most recent files are first.

`file-newest-backup` *filename* [Function]

This function returns the first element of the list returned by `file-backup-file-names`.

Some file comparison commands use this function so that they can automatically compare a file with its most recent backup.

27.2 Auto-Saving

Emacs periodically saves all files that you are visiting; this is called *auto-saving*. Auto-saving prevents you from losing more than a limited amount of work if the system crashes. By default, auto-saves happen every 300 keystrokes, or after around 30 seconds of idle time. See Section "Auto-Saving: Protection Against Disasters" in *The GNU Emacs Manual*, for information on auto-save for users. Here we describe the functions used to implement auto-saving and the variables that control them.

`buffer-auto-save-file-name` [Variable]

This buffer-local variable is the name of the file used for auto-saving the current buffer. It is `nil` if the buffer should not be auto-saved.

```
buffer-auto-save-file-name
     ⇒ "/xcssun/users/rms/lewis/#backups.texi#"
```

`auto-save-mode` *arg* [Command]

This is the mode command for Auto Save mode, a buffer-local minor mode. When Auto Save mode is enabled, auto-saving is enabled in the buffer. The calling convention is the same as for other minor mode commands (see Section 24.3.1 [Minor Mode Conventions], page 545).

Unlike most minor modes, there is no `auto-save-mode` variable. Auto Save mode is enabled if `buffer-auto-save-file-name` is non-`nil` and `buffer-saved-size` (see below) is non-zero.

`auto-save-file-name-transforms` [Variable]

This variable lists transforms to apply to buffer's file name before making the auto-save file name.

Each transform is a list of the form `(`*`regexp replacement`* `[`*`uniquify`*`])`. *regexp* is a regular expression to match against the file name; if it matches, `replace-match` is used to replace the matching part with *replacement*. If the optional element *uniquify* is non-`nil`, the auto-save file name is constructed by concatenating the directory part of the transformed file name with the buffer's file name in which all directory separators were changed to '`!`' to prevent clashes. (This will not work correctly if your filesystem truncates the resulting name.)

If *uniquify* is one of the members of `secure-hash-algorithms`, Emacs constructs the nondirectory part of the auto-save file name by applying that `secure-hash` to the buffer file name. This avoids any risk of excessively long file names.

All the transforms in the list are tried, in the order they are listed. When one transform applies, its result is final; no further transforms are tried.

The default value is set up to put the auto-save files of remote files into the temporary directory (see Section 26.9.5 [Unique File Names], page 644).

On MS-DOS filesystems without long names this variable is always ignored.

`auto-save-file-name-p` *filename* [Function]

This function returns a non-`nil` value if *filename* is a string that could be the name of an auto-save file. It assumes the usual naming convention for auto-save files: a name that begins and ends with hash marks ('`#`') is a possible auto-save file name. The argument *filename* should not contain a directory part.

```
(make-auto-save-file-name)
     ⇒ "/xcssun/users/rms/lewis/#backups.texi#"
(auto-save-file-name-p "#backups.texi#")
     ⇒ 0
(auto-save-file-name-p "backups.texi")
     ⇒ nil
```

`make-auto-save-file-name` [Function]

This function returns the file name to use for auto-saving the current buffer. This is just the file name with hash marks ('`#`') prepended and appended to it. This function does not look at the variable `auto-save-visited-file-name` (described below); callers of this function should check that variable first.

```
(make-auto-save-file-name)
     ⇒ "/xcssun/users/rms/lewis/#backups.texi#"
```

`auto-save-visited-file-name` [User Option]

If this variable is non-`nil`, Emacs auto-saves buffers in the files they are visiting. That is, the auto-save is done in the same file that you are editing. Normally, this variable is `nil`, so auto-save files have distinct names that are created by `make-auto-save-file-name`.

When you change the value of this variable, the new value does not take effect in an existing buffer until the next time auto-save mode is reenabled in it. If auto-save mode is already enabled, auto-saves continue to go in the same file name until `auto-save-mode` is called again.

Note that setting this variable to a non-`nil` value does not change the fact that auto-saving is different from saving the buffer; e.g., the hooks described in Section 26.2 [Saving Buffers], page 614, are *not* run when a buffer is auto-saved.

`recent-auto-save-p` [Function]
This function returns `t` if the current buffer has been auto-saved since the last time it was read in or saved.

`set-buffer-auto-saved` [Function]
This function marks the current buffer as auto-saved. The buffer will not be auto-saved again until the buffer text is changed again. The function returns `nil`.

`auto-save-interval` [User Option]
The value of this variable specifies how often to do auto-saving, in terms of number of input events. Each time this many additional input events are read, Emacs does auto-saving for all buffers in which that is enabled. Setting this to zero disables autosaving based on the number of characters typed.

`auto-save-timeout` [User Option]
The value of this variable is the number of seconds of idle time that should cause auto-saving. Each time the user pauses for this long, Emacs does auto-saving for all buffers in which that is enabled. (If the current buffer is large, the specified timeout is multiplied by a factor that increases as the size increases; for a million-byte buffer, the factor is almost 4.)

If the value is zero or `nil`, then auto-saving is not done as a result of idleness, only after a certain number of input events as specified by `auto-save-interval`.

`auto-save-hook` [Variable]
This normal hook is run whenever an auto-save is about to happen.

`auto-save-default` [User Option]
If this variable is non-`nil`, buffers that are visiting files have auto-saving enabled by default. Otherwise, they do not.

`do-auto-save` **&optional** *no-message current-only* [Command]
This function auto-saves all buffers that need to be auto-saved. It saves all buffers for which auto-saving is enabled and that have been changed since the previous auto-save.

If any buffers are auto-saved, `do-auto-save` normally displays a message saying '`Auto-saving...`' in the echo area while auto-saving is going on. However, if *no-message* is non-`nil`, the message is inhibited.

If *current-only* is non-`nil`, only the current buffer is auto-saved.

`delete-auto-save-file-if-necessary` **&optional** *force* [Function]
This function deletes the current buffer's auto-save file if `delete-auto-save-files` is non-`nil`. It is called every time a buffer is saved.

Unless *force* is non-`nil`, this function only deletes the file if it was written by the current Emacs session since the last true save.

`delete-auto-save-files` [User Option]
: This variable is used by the function `delete-auto-save-file-if-necessary`. If it is non-`nil`, Emacs deletes auto-save files when a true save is done (in the visited file). This saves disk space and unclutters your directory.

`rename-auto-save-file` [Function]
: This function adjusts the current buffer's auto-save file name if the visited file name has changed. It also renames an existing auto-save file, if it was made in the current Emacs session. If the visited file name has not changed, this function does nothing.

`buffer-saved-size` [Variable]
: The value of this buffer-local variable is the length of the current buffer, when it was last read in, saved, or auto-saved. This is used to detect a substantial decrease in size, and turn off auto-saving in response.

  If it is -1, that means auto-saving is temporarily shut off in this buffer due to a substantial decrease in size. Explicitly saving the buffer stores a positive value in this variable, thus reenabling auto-saving. Turning auto-save mode off or on also updates this variable, so that the substantial decrease in size is forgotten.

  If it is -2, that means this buffer should disregard changes in buffer size; in particular, it should not shut off auto-saving temporarily due to changes in buffer size.

`auto-save-list-file-name` [Variable]
: This variable (if non-`nil`) specifies a file for recording the names of all the auto-save files. Each time Emacs does auto-saving, it writes two lines into this file for each buffer that has auto-saving enabled. The first line gives the name of the visited file (it's empty if the buffer has none), and the second gives the name of the auto-save file.

  When Emacs exits normally, it deletes this file; if Emacs crashes, you can look in the file to find all the auto-save files that might contain work that was otherwise lost. The `recover-session` command uses this file to find them.

  The default name for this file specifies your home directory and starts with '`.saves-`'. It also contains the Emacs process ID and the host name.

`auto-save-list-file-prefix` [User Option]
: After Emacs reads your init file, it initializes `auto-save-list-file-name` (if you have not already set it non-`nil`) based on this prefix, adding the host name and process ID. If you set this to `nil` in your init file, then Emacs does not initialize `auto-save-list-file-name`.

27.3 Reverting

If you have made extensive changes to a file and then change your mind about them, you can get rid of them by reading in the previous version of the file with the `revert-buffer` command. See Section "Reverting a Buffer" in *The GNU Emacs Manual*.

`revert-buffer` **&optional** *ignore-auto noconfirm preserve-modes* [Command]
: This command replaces the buffer text with the text of the visited file on disk. This action undoes all changes since the file was visited or saved.

By default, if the latest auto-save file is more recent than the visited file, and the argument *ignore-auto* is `nil`, `revert-buffer` asks the user whether to use that auto-save instead. When you invoke this command interactively, *ignore-auto* is `t` if there is no numeric prefix argument; thus, the interactive default is not to check the auto-save file.

Normally, `revert-buffer` asks for confirmation before it changes the buffer; but if the argument *noconfirm* is non-`nil`, `revert-buffer` does not ask for confirmation.

Normally, this command reinitializes the buffer's major and minor modes using `normal-mode`. But if *preserve-modes* is non-`nil`, the modes remain unchanged.

Reverting tries to preserve marker positions in the buffer by using the replacement feature of `insert-file-contents`. If the buffer contents and the file contents are identical before the revert operation, reverting preserves all the markers. If they are not identical, reverting does change the buffer; in that case, it preserves the markers in the unchanged text (if any) at the beginning and end of the buffer. Preserving any additional markers would be problematic.

When reverting from non-file sources, markers are usually not preserved, but this is up to the specific `revert-buffer-function` implementation.

`revert-buffer-in-progress-p` [Variable]

`revert-buffer` binds this variable to a non-`nil` value while it is working.

You can customize how `revert-buffer` does its work by setting the variables described in the rest of this section.

`revert-without-query` [User Option]

This variable holds a list of files that should be reverted without query. The value is a list of regular expressions. If the visited file name matches one of these regular expressions, and the file has changed on disk but the buffer is not modified, then `revert-buffer` reverts the file without asking the user for confirmation.

Some major modes customize `revert-buffer` by making buffer-local bindings for these variables:

`revert-buffer-function` [Variable]

The value of this variable is the function to use to revert this buffer. It should be a function with two optional arguments to do the work of reverting. The two optional arguments, *ignore-auto* and *noconfirm*, are the arguments that `revert-buffer` received.

Modes such as Dired mode, in which the text being edited does not consist of a file's contents but can be regenerated in some other fashion, can give this variable a buffer-local value that is a special function to regenerate the contents.

`revert-buffer-insert-file-contents-function` [Variable]

The value of this variable specifies the function to use to insert the updated contents when reverting this buffer. The function receives two arguments: first the file name to use; second, `t` if the user has asked to read the auto-save file.

The reason for a mode to change this variable instead of `revert-buffer-function` is to avoid duplicating or replacing the rest of what `revert-buffer` does: asking for

confirmation, clearing the undo list, deciding the proper major mode, and running the hooks listed below.

`before-revert-hook` [Variable]
: This normal hook is run by the default `revert-buffer-function` before inserting the modified contents. A custom `revert-buffer-function` may or may not run this hook.

`after-revert-hook` [Variable]
: This normal hook is run by the default `revert-buffer-function` after inserting the modified contents. A custom `revert-buffer-function` may or may not run this hook.

Emacs can revert buffers automatically. It does that by default for buffers visiting files. The following describes how to add support for auto-reverting new types of buffers.

First, such buffers must have a suitable `revert-buffer-function` and `buffer-stale-function` defined.

`buffer-stale-function` [Variable]
: The value of this variable specifies a function to call to check whether a buffer needs reverting. The default value only handles buffers that are visiting files, by checking their modification time. Buffers that are not visiting files require a custom function of one optional argument *noconfirm*. The function should return non-`nil` if the buffer should be reverted. The buffer is current when this function is called.

  While this function is mainly intended for use in auto-reverting, it could be used for other purposes as well. For instance, if auto-reverting is not enabled, it could be used to warn the user that the buffer needs reverting. The idea behind the *noconfirm* argument is that it should be `t` if the buffer is going to be reverted without asking the user and `nil` if the function is just going to be used to warn the user that the buffer is out of date. In particular, for use in auto-reverting, *noconfirm* is `t`. If the function is only going to be used for auto-reverting, you can ignore the *noconfirm* argument.

  If you just want to automatically auto-revert every `auto-revert-interval` seconds (like the Buffer Menu), use:

```
(setq-local buffer-stale-function
     (lambda (&optional noconfirm) 'fast))
```

  in the buffer's mode function.

  The special return value '`fast`' tells the caller that the need for reverting was not checked, but that reverting the buffer is fast. It also tells Auto Revert not to print any revert messages, even if `auto-revert-verbose` is non-`nil`. This is important, as getting revert messages every `auto-revert-interval` seconds can be very annoying. The information provided by this return value could also be useful if the function is consulted for purposes other than auto-reverting.

Once the buffer has a suitable `revert-buffer-function` and `buffer-stale-function`, several problems usually remain.

The buffer will only auto-revert if it is marked unmodified. Hence, you will have to make sure that various functions mark the buffer modified if and only if either the buffer contains

information that might be lost by reverting, or there is reason to believe that the user might be inconvenienced by auto-reverting, because he is actively working on the buffer. The user can always override this by manually adjusting the modified status of the buffer. To support this, calling the `revert-buffer-function` on a buffer that is marked unmodified should always keep the buffer marked unmodified.

It is important to assure that point does not continuously jump around as a consequence of auto-reverting. Of course, moving point might be inevitable if the buffer radically changes.

You should make sure that the `revert-buffer-function` does not print messages that unnecessarily duplicate Auto Revert's own messages, displayed if `auto-revert-verbose` is `t`, and effectively override a `nil` value for `auto-revert-verbose`. Hence, adapting a mode for auto-reverting often involves getting rid of such messages. This is especially important for buffers that automatically revert every `auto-revert-interval` seconds.

If the new auto-reverting is part of Emacs, you should mention it in the documentation string of `global-auto-revert-non-file-buffers`.

Similarly, you should document the additions in the Emacs manual.

28 Buffers

A *buffer* is a Lisp object containing text to be edited. Buffers are used to hold the contents of files that are being visited; there may also be buffers that are not visiting files. While several buffers may exist at one time, only one buffer is designated the *current buffer* at any time. Most editing commands act on the contents of the current buffer. Each buffer, including the current buffer, may or may not be displayed in any windows.

28.1 Buffer Basics

Buffers in Emacs editing are objects that have distinct names and hold text that can be edited. Buffers appear to Lisp programs as a special data type. You can think of the contents of a buffer as a string that you can extend; insertions and deletions may occur in any part of the buffer. See Chapter 33 [Text], page 880.

A Lisp buffer object contains numerous pieces of information. Some of this information is directly accessible to the programmer through variables, while other information is accessible only through special-purpose functions. For example, the visited file name is directly accessible through a variable, while the value of point is accessible only through a primitive function.

Buffer-specific information that is directly accessible is stored in *buffer-local* variable bindings, which are variable values that are effective only in a particular buffer. This feature allows each buffer to override the values of certain variables. Most major modes override variables such as `fill-column` or `comment-column` in this way. For more information about buffer-local variables and functions related to them, see Section 12.11 [Buffer-Local Variables], page 207.

For functions and variables related to visiting files in buffers, see Section 26.1 [Visiting Files], page 610, and Section 26.2 [Saving Buffers], page 614. For functions and variables related to the display of buffers in windows, see Section 29.11 [Buffers and Windows], page 722.

`bufferp` *object* [Function]
: This function returns `t` if *object* is a buffer, `nil` otherwise.

28.2 The Current Buffer

There are, in general, many buffers in an Emacs session. At any time, one of them is designated the *current buffer*—the buffer in which most editing takes place. Most of the primitives for examining or changing text operate implicitly on the current buffer (see Chapter 33 [Text], page 880).

Normally, the buffer displayed in the selected window (see Section 29.3 [Selecting Windows], page 698) is the current buffer, but this is not always so: a Lisp program can temporarily designate any buffer as current in order to operate on its contents, without changing what is displayed on the screen. The most basic function for designating a current buffer is `set-buffer`.

`current-buffer` [Function]
: This function returns the current buffer.

```
(current-buffer)
     ⇒ #<buffer buffers.texi>
```

`set-buffer` *buffer-or-name* [Function]

This function makes *buffer-or-name* the current buffer. *buffer-or-name* must be an existing buffer or the name of an existing buffer. The return value is the buffer made current.

This function does not display the buffer in any window, so the user cannot necessarily see the buffer. But Lisp programs will now operate on it.

When an editing command returns to the editor command loop, Emacs automatically calls `set-buffer` on the buffer shown in the selected window (see Section 29.3 [Selecting Windows], page 698). This is to prevent confusion: it ensures that the buffer that the cursor is in, when Emacs reads a command, is the buffer to which that command applies (see Chapter 22 [Command Loop], page 424). Thus, you should not use `set-buffer` to switch visibly to a different buffer; for that, use the functions described in Section 29.12 [Switching Buffers], page 723.

When writing a Lisp function, do *not* rely on this behavior of the command loop to restore the current buffer after an operation. Editing commands can also be called as Lisp functions by other programs, not just from the command loop; it is convenient for the caller if the subroutine does not change which buffer is current (unless, of course, that is the subroutine's purpose).

To operate temporarily on another buffer, put the `set-buffer` within a `save-current-buffer` form. Here, as an example, is a simplified version of the command `append-to-buffer`:

```
(defun append-to-buffer (buffer start end)
  "Append the text of the region to BUFFER."
  (interactive "BAppend to buffer: \nr")
  (let ((oldbuf (current-buffer)))
    (save-current-buffer
      (set-buffer (get-buffer-create buffer))
      (insert-buffer-substring oldbuf start end))))
```

Here, we bind a local variable to record the current buffer, and then `save-current-buffer` arranges to make it current again later. Next, `set-buffer` makes the specified buffer current, and `insert-buffer-substring` copies the string from the original buffer to the specified (and now current) buffer.

Alternatively, we can use the `with-current-buffer` macro:

```
(defun append-to-buffer (buffer start end)
  "Append the text of the region to BUFFER."
  (interactive "BAppend to buffer: \nr")
  (let ((oldbuf (current-buffer)))
    (with-current-buffer (get-buffer-create buffer)
      (insert-buffer-substring oldbuf start end))))
```

In either case, if the buffer appended to happens to be displayed in some window, the next redisplay will show how its text has changed. If it is not displayed in any window,

you will not see the change immediately on the screen. The command causes the buffer to become current temporarily, but does not cause it to be displayed.

If you make local bindings (with `let` or function arguments) for a variable that may also have buffer-local bindings, make sure that the same buffer is current at the beginning and at the end of the local binding's scope. Otherwise you might bind it in one buffer and unbind it in another!

Do not rely on using `set-buffer` to change the current buffer back, because that won't do the job if a quit happens while the wrong buffer is current. For instance, in the previous example, it would have been wrong to do this:

```
(let ((oldbuf (current-buffer)))
  (set-buffer (get-buffer-create buffer))
  (insert-buffer-substring oldbuf start end)
  (set-buffer oldbuf))
```

Using `save-current-buffer` or `with-current-buffer`, as we did, correctly handles quitting, errors, and `throw`, as well as ordinary evaluation.

`save-current-buffer` *body...* [Special Form]

The `save-current-buffer` special form saves the identity of the current buffer, evaluates the *body* forms, and finally restores that buffer as current. The return value is the value of the last form in *body*. The current buffer is restored even in case of an abnormal exit via `throw` or error (see Section 11.7 [Nonlocal Exits], page 177).

If the buffer that used to be current has been killed by the time of exit from `save-current-buffer`, then it is not made current again, of course. Instead, whichever buffer was current just before exit remains current.

`with-current-buffer` *buffer-or-name body...* [Macro]

The `with-current-buffer` macro saves the identity of the current buffer, makes *buffer-or-name* current, evaluates the *body* forms, and finally restores the current buffer. *buffer-or-name* must specify an existing buffer or the name of an existing buffer.

The return value is the value of the last form in *body*. The current buffer is restored even in case of an abnormal exit via `throw` or error (see Section 11.7 [Nonlocal Exits], page 177).

`with-temp-buffer` *body...* [Macro]

The `with-temp-buffer` macro evaluates the *body* forms with a temporary buffer as the current buffer. It saves the identity of the current buffer, creates a temporary buffer and makes it current, evaluates the *body* forms, and finally restores the previous current buffer while killing the temporary buffer.

By default, undo information (see Section 33.9 [Undo], page 898) is not recorded in the buffer created by this macro (but *body* can enable that, if needed). The temporary buffer also does not run the hooks `kill-buffer-hook`, `kill-buffer-query-functions` (see Section 28.10 [Killing Buffers], page 687), and `buffer-list-update-hook` (see Section 28.8 [Buffer List], page 683).

The return value is the value of the last form in *body*. You can return the contents of the temporary buffer by using `(buffer-string)` as the last form.

The current buffer is restored even in case of an abnormal exit via `throw` or error (see Section 11.7 [Nonlocal Exits], page 177).

See also `with-temp-file` in [Writing to Files], page 619.

28.3 Buffer Names

Each buffer has a unique name, which is a string. Many of the functions that work on buffers accept either a buffer or a buffer name as an argument. Any argument called *buffer-or-name* is of this sort, and an error is signaled if it is neither a string nor a buffer. Any argument called *buffer* must be an actual buffer object, not a name.

Buffers that are ephemeral and generally uninteresting to the user have names starting with a space, so that the `list-buffers` and `buffer-menu` commands don't mention them (but if such a buffer visits a file, it **is** mentioned). A name starting with space also initially disables recording undo information; see Section 33.9 [Undo], page 898.

`buffer-name` **&optional** *buffer* [Function]

This function returns the name of *buffer* as a string. *buffer* defaults to the current buffer.

If `buffer-name` returns `nil`, it means that *buffer* has been killed. See Section 28.10 [Killing Buffers], page 687.

```
(buffer-name)
     ⇒ "buffers.texi"

(setq foo (get-buffer "temp"))
     ⇒ #<buffer temp>
(kill-buffer foo)
     ⇒ nil
(buffer-name foo)
     ⇒ nil
foo
     ⇒ #<killed buffer>
```

`rename-buffer` *newname* **&optional** *unique* [Command]

This function renames the current buffer to *newname*. An error is signaled if *newname* is not a string.

Ordinarily, `rename-buffer` signals an error if *newname* is already in use. However, if *unique* is non-`nil`, it modifies *newname* to make a name that is not in use. Interactively, you can make *unique* non-`nil` with a numeric prefix argument. (This is how the command `rename-uniquely` is implemented.)

This function returns the name actually given to the buffer.

`get-buffer` *buffer-or-name* [Function]

This function returns the buffer specified by *buffer-or-name*. If *buffer-or-name* is a string and there is no buffer with that name, the value is `nil`. If *buffer-or-name* is a buffer, it is returned as given; that is not very useful, so the argument is usually a name. For example:

```
(setq b (get-buffer "lewis"))
     ⇒ #<buffer lewis>
```

```
(get-buffer b)
     ⇒ #<buffer lewis>
(get-buffer "Frazzle-nots")
     ⇒ nil
```

See also the function `get-buffer-create` in Section 28.9 [Creating Buffers], page 686.

`generate-new-buffer-name` *starting-name* **&optional** *ignore* [Function]

This function returns a name that would be unique for a new buffer—but does not create the buffer. It starts with *starting-name*, and produces a name not currently in use for any buffer by appending a number inside of '`<...>`'. It starts at 2 and keeps incrementing the number until it is not the name of an existing buffer.

If the optional second argument *ignore* is non-`nil`, it should be a string, a potential buffer name. It means to consider that potential buffer acceptable, if it is tried, even if it is the name of an existing buffer (which would normally be rejected). Thus, if buffers named '`foo`', '`foo<2>`', '`foo<3>`' and '`foo<4>`' exist,

```
(generate-new-buffer-name "foo")
     ⇒ "foo<5>"
(generate-new-buffer-name "foo" "foo<3>")
     ⇒ "foo<3>"
(generate-new-buffer-name "foo" "foo<6>")
     ⇒ "foo<5>"
```

See the related function `generate-new-buffer` in Section 28.9 [Creating Buffers], page 686.

28.4 Buffer File Name

The *buffer file name* is the name of the file that is visited in that buffer. When a buffer is not visiting a file, its buffer file name is `nil`. Most of the time, the buffer name is the same as the nondirectory part of the buffer file name, but the buffer file name and the buffer name are distinct and can be set independently. See Section 26.1 [Visiting Files], page 610.

`buffer-file-name` **&optional** *buffer* [Function]

This function returns the absolute file name of the file that *buffer* is visiting. If *buffer* is not visiting any file, `buffer-file-name` returns `nil`. If *buffer* is not supplied, it defaults to the current buffer.

```
(buffer-file-name (other-buffer))
     ⇒ "/usr/user/lewis/manual/files.texi"
```

`buffer-file-name` [Variable]

This buffer-local variable contains the name of the file being visited in the current buffer, or `nil` if it is not visiting a file. It is a permanent local variable, unaffected by `kill-all-local-variables`.

```
buffer-file-name
     ⇒ "/usr/user/lewis/manual/buffers.texi"
```

It is risky to change this variable's value without doing various other things. Normally it is better to use `set-visited-file-name` (see below); some of the things done there, such as changing the buffer name, are not strictly necessary, but others are essential to avoid confusing Emacs.

`buffer-file-truename` [Variable]
This buffer-local variable holds the abbreviated truename of the file visited in the current buffer, or `nil` if no file is visited. It is a permanent local, unaffected by `kill-all-local-variables`. See Section 26.6.3 [Truenames], page 625, and [abbreviate-file-name], page 641.

`buffer-file-number` [Variable]
This buffer-local variable holds the inode number and device identifier of the file visited in the current buffer, or `nil` if no file or a nonexistent file is visited. It is a permanent local, unaffected by `kill-all-local-variables`.

The value is normally a list of the form `(`*`inodenum device`*`)`. This tuple uniquely identifies the file among all files accessible on the system. See the function `file-attributes`, in Section 26.6.4 [File Attributes], page 626, for more information about them.

If `buffer-file-name` is the name of a symbolic link, then both *inodenum* and *device* refer to the recursive target of the link.

`get-file-buffer` *filename* [Function]
This function returns the buffer visiting file *filename*. If there is no such buffer, it returns `nil`. The argument *filename*, which must be a string, is expanded (see Section 26.9.4 [File Name Expansion], page 641), then compared against the visited file names of all live buffers. Note that the buffer's `buffer-file-name` must match the expansion of *filename* exactly. This function will not recognize other names for the same file.

```
(get-file-buffer "buffers.texi")
    ⇒ #<buffer buffers.texi>
```

In unusual circumstances, there can be more than one buffer visiting the same file name. In such cases, this function returns the first such buffer in the buffer list.

`find-buffer-visiting` *filename* **&optional** *predicate* [Function]
This is like `get-file-buffer`, except that it can return any buffer visiting the file *possibly under a different name*. That is, the buffer's `buffer-file-name` does not need to match the expansion of *filename* exactly, it only needs to refer to the same file. If *predicate* is non-`nil`, it should be a function of one argument, a buffer visiting *filename*. The buffer is only considered a suitable return value if *predicate* returns non-`nil`. If it can not find a suitable buffer to return, `find-buffer-visiting` returns `nil`.

`set-visited-file-name` *filename* **&optional** *no-query along-with-file* [Command]
If *filename* is a non-empty string, this function changes the name of the file visited in the current buffer to *filename*. (If the buffer had no visited file, this gives it one.) The *next time* the buffer is saved it will go in the newly-specified file.

This command marks the buffer as modified, since it does not (as far as Emacs knows) match the contents of *filename*, even if it matched the former visited file. It also renames the buffer to correspond to the new file name, unless the new name is already in use.

If *filename* is `nil` or the empty string, that stands for "no visited file". In this case, `set-visited-file-name` marks the buffer as having no visited file, without changing the buffer's modified flag.

Normally, this function asks the user for confirmation if there already is a buffer visiting *filename*. If *no-query* is non-`nil`, that prevents asking this question. If there already is a buffer visiting *filename*, and the user confirms or *no-query* is non-`nil`, this function makes the new buffer name unique by appending a number inside of '`<...>`' to *filename*.

If *along-with-file* is non-`nil`, that means to assume that the former visited file has been renamed to *filename*. In this case, the command does not change the buffer's modified flag, nor the buffer's recorded last file modification time as reported by `visited-file-modtime` (see Section 28.6 [Modification Time], page 680). If *along-with-file* is `nil`, this function clears the recorded last file modification time, after which `visited-file-modtime` returns zero.

When the function `set-visited-file-name` is called interactively, it prompts for *filename* in the minibuffer.

`list-buffers-directory` [Variable]

This buffer-local variable specifies a string to display in a buffer listing where the visited file name would go, for buffers that don't have a visited file name. Dired buffers use this variable.

28.5 Buffer Modification

Emacs keeps a flag called the *modified flag* for each buffer, to record whether you have changed the text of the buffer. This flag is set to `t` whenever you alter the contents of the buffer, and cleared to `nil` when you save it. Thus, the flag shows whether there are unsaved changes. The flag value is normally shown in the mode line (see Section 24.4.4 [Mode Line Variables], page 555), and controls saving (see Section 26.2 [Saving Buffers], page 614) and auto-saving (see Section 27.2 [Auto-Saving], page 666).

Some Lisp programs set the flag explicitly. For example, the function `set-visited-file-name` sets the flag to `t`, because the text does not match the newly-visited file, even if it is unchanged from the file formerly visited.

The functions that modify the contents of buffers are described in Chapter 33 [Text], page 880.

`buffer-modified-p` **&optional** *buffer* [Function]

This function returns non-`nil` if *buffer* has been modified since it was last read in from a file or saved, or `nil` otherwise. If *buffer* has been auto-saved since the time it was last modified, this function returns the symbol `autosaved`. If *buffer* is `nil` or omitted, it defaults to the current buffer.

`set-buffer-modified-p` *flag* [Function]

This function marks the current buffer as modified if *flag* is non-`nil`, or as unmodified if the flag is `nil`.

Another effect of calling this function is to cause unconditional redisplay of the mode line for the current buffer. In fact, the function `force-mode-line-update` works by doing this:

```
(set-buffer-modified-p (buffer-modified-p))
```

`restore-buffer-modified-p` *flag* [Function]
Like `set-buffer-modified-p`, but does not force redisplay of mode lines. This function also allows *flag*'s value to be the symbol `autosaved`, which marks the buffer as modified and auto-saved after the last modification.

`not-modified` **&optional** *arg* [Command]
This command marks the current buffer as unmodified, and not needing to be saved. If *arg* is non-`nil`, it marks the buffer as modified, so that it will be saved at the next suitable occasion. Interactively, *arg* is the prefix argument.

Don't use this function in programs, since it prints a message in the echo area; use `set-buffer-modified-p` (above) instead.

`buffer-modified-tick` **&optional** *buffer* [Function]
This function returns *buffer*'s modification-count. This is a counter that increments every time the buffer is modified. If *buffer* is `nil` (or omitted), the current buffer is used.

`buffer-chars-modified-tick` **&optional** *buffer* [Function]
This function returns *buffer*'s character-change modification-count. Changes to text properties leave this counter unchanged; however, each time text is inserted or removed from the buffer, the counter is reset to the value that would be returned by `buffer-modified-tick`. By comparing the values returned by two `buffer-chars-modified-tick` calls, you can tell whether a character change occurred in that buffer in between the calls. If *buffer* is `nil` (or omitted), the current buffer is used.

Sometimes there's a need for modifying buffer in a way that doesn't really change its text, like if only its text properties are changed. If your program needs to modify a buffer without triggering any hooks and features that react to buffer modifications, use the `with-silent-modifications` macro.

`with-silent-modifications` *body...* [Macro]
Execute *body* pretending it does not modify the buffer. This includes checking whether the buffer's file is locked (see Section 26.5 [File Locks], page 619), running buffer modification hooks (see Section 33.34 [Change Hooks], page 964), etc. Note that if *body* actually modifies the buffer text (as opposed to its text properties), its undo data may become corrupted.

28.6 Buffer Modification Time

Suppose that you visit a file and make changes in its buffer, and meanwhile the file itself is changed on disk. At this point, saving the buffer would overwrite the changes in the file. Occasionally this may be what you want, but usually it would lose valuable information. Emacs therefore checks the file's modification time using the functions described below before saving the file. (See Section 26.6.4 [File Attributes], page 626, for how to examine a file's modification time.)

`verify-visited-file-modtime` **&optional** *buffer* [Function]
This function compares what *buffer* (by default, the current-buffer) has recorded for the modification time of its visited file against the actual modification time of the file as recorded by the operating system. The two should be the same unless some other process has written the file since Emacs visited or saved it.

The function returns `t` if the last actual modification time and Emacs's recorded modification time are the same, `nil` otherwise. It also returns `t` if the buffer has no recorded last modification time, that is if `visited-file-modtime` would return zero.

It always returns `t` for buffers that are not visiting a file, even if `visited-file-modtime` returns a non-zero value. For instance, it always returns `t` for dired buffers. It returns `t` for buffers that are visiting a file that does not exist and never existed, but `nil` for file-visiting buffers whose file has been deleted.

`clear-visited-file-modtime` [Function]
This function clears out the record of the last modification time of the file being visited by the current buffer. As a result, the next attempt to save this buffer will not complain of a discrepancy in file modification times.

This function is called in `set-visited-file-name` and other exceptional places where the usual test to avoid overwriting a changed file should not be done.

`visited-file-modtime` [Function]
This function returns the current buffer's recorded last file modification time, as a Lisp timestamp (see Section 42.5 [Time of Day], page 1275).

If the buffer has no recorded last modification time, this function returns zero. This case occurs, for instance, if the buffer is not visiting a file or if the time has been explicitly cleared by `clear-visited-file-modtime`. Note, however, that `visited-file-modtime` returns a timestamp for some non-file buffers too. For instance, in a Dired buffer listing a directory, it returns the last modification time of that directory, as recorded by Dired.

If the buffer is visiting a file that doesn't exist, this function returns −1.

`set-visited-file-modtime` **&optional** *time* [Function]
This function updates the buffer's record of the last modification time of the visited file, to the value specified by *time* if *time* is not `nil`, and otherwise to the last modification time of the visited file.

If *time* is neither `nil` nor an integer flag returned by `visited-file-modtime`, it should be a Lisp time value (see Section 42.5 [Time of Day], page 1275).

This function is useful if the buffer was not read from the file normally, or if the file itself has been changed for some known benign reason.

`ask-user-about-supersession-threat` *filename* [Function]
This function is used to ask a user how to proceed after an attempt to modify a buffer visiting file *filename* when the file is newer than the buffer text. Emacs detects this because the modification time of the file on disk is newer than the last save-time and its contents have changed. This means some other program has probably altered the file.

Depending on the user's answer, the function may return normally, in which case the modification of the buffer proceeds, or it may signal a `file-supersession` error with data `(filename)`, in which case the proposed buffer modification is not allowed.

This function is called automatically by Emacs on the proper occasions. It exists so you can customize Emacs by redefining it. See the file `userlock.el` for the standard definition.

See also the file locking mechanism in Section 26.5 [File Locks], page 619.

28.7 Read-Only Buffers

If a buffer is *read-only*, then you cannot change its contents, although you may change your view of the contents by scrolling and narrowing.

Read-only buffers are used in two kinds of situations:

- A buffer visiting a write-protected file is normally read-only.

  Here, the purpose is to inform the user that editing the buffer with the aim of saving it in the file may be futile or undesirable. The user who wants to change the buffer text despite this can do so after clearing the read-only flag with `C-x C-q`.

- Modes such as Dired and Rmail make buffers read-only when altering the contents with the usual editing commands would probably be a mistake.

  The special commands of these modes bind `buffer-read-only` to `nil` (with `let`) or bind `inhibit-read-only` to `t` around the places where they themselves change the text.

`buffer-read-only` [Variable]

This buffer-local variable specifies whether the buffer is read-only. The buffer is read-only if this variable is non-`nil`. However, characters that have the `inhibit-read-only` text property can still be modified. See Section 33.19.4 [Special Properties], page 927.

`inhibit-read-only` [Variable]

If this variable is non-`nil`, then read-only buffers and, depending on the actual value, some or all read-only characters may be modified. Read-only characters in a buffer are those that have a non-`nil` `read-only` text property. See Section 33.19.4 [Special Properties], page 927, for more information about text properties.

If `inhibit-read-only` is `t`, all `read-only` character properties have no effect. If `inhibit-read-only` is a list, then `read-only` character properties have no effect if they are members of the list (comparison is done with `eq`).

`read-only-mode` **&optional** *arg* [Command]

This is the mode command for Read Only minor mode, a buffer-local minor mode. When the mode is enabled, `buffer-read-only` is non-`nil` in the buffer; when disabled, `buffer-read-only` is `nil` in the buffer. The calling convention is the same as for other minor mode commands (see Section 24.3.1 [Minor Mode Conventions], page 545).

This minor mode mainly serves as a wrapper for `buffer-read-only`; unlike most minor modes, there is no separate `read-only-mode` variable. Even when Read Only

mode is disabled, characters with non-`nil` `read-only` text properties remain read-only. To temporarily ignore all read-only states, bind `inhibit-read-only`, as described above.

When enabling Read Only mode, this mode command also enables View mode if the option `view-read-only` is non-`nil`. See Section "Miscellaneous Buffer Operations" in *The GNU Emacs Manual*. When disabling Read Only mode, it disables View mode if View mode was enabled.

`barf-if-buffer-read-only` **&optional** *position* [Function]

This function signals a `buffer-read-only` error if the current buffer is read-only. If the text at *position* (which defaults to point) has the `inhibit-read-only` text property set, the error will not be raised.

See Section 22.2.1 [Using Interactive], page 425, for another way to signal an error if the current buffer is read-only.

28.8 The Buffer List

The *buffer list* is a list of all live buffers. The order of the buffers in this list is based primarily on how recently each buffer has been displayed in a window. Several functions, notably `other-buffer`, use this ordering. A buffer list displayed for the user also follows this order.

Creating a buffer adds it to the end of the buffer list, and killing a buffer removes it from that list. A buffer moves to the front of this list whenever it is chosen for display in a window (see Section 29.12 [Switching Buffers], page 723) or a window displaying it is selected (see Section 29.3 [Selecting Windows], page 698). A buffer moves to the end of the list when it is buried (see `bury-buffer`, below). There are no functions available to the Lisp programmer which directly manipulate the buffer list.

In addition to the fundamental buffer list just described, Emacs maintains a local buffer list for each frame, in which the buffers that have been displayed (or had their windows selected) in that frame come first. (This order is recorded in the frame's `buffer-list` frame parameter; see Section 30.4.3.5 [Buffer Parameters], page 814.) Buffers never displayed in that frame come afterward, ordered according to the fundamental buffer list.

`buffer-list` **&optional** *frame* [Function]

This function returns the buffer list, including all buffers, even those whose names begin with a space. The elements are actual buffers, not their names.

If *frame* is a frame, this returns *frame*'s local buffer list. If *frame* is `nil` or omitted, the fundamental buffer list is used: the buffers appear in order of most recent display or selection, regardless of which frames they were displayed on.

```
(buffer-list)
     ⇒ (#<buffer buffers.texi>
         #<buffer  *Minibuf-1*> #<buffer buffer.c>
         #<buffer *Help*> #<buffer TAGS>)
```

```
;; Note that the name of the minibuffer
;;    begins with a space!
(mapcar #'buffer-name (buffer-list))
    ⇒ ("buffers.texi" " *Minibuf-1*"
       "buffer.c" "*Help*" "TAGS")
```

The list returned by `buffer-list` is constructed specifically; it is not an internal Emacs data structure, and modifying it has no effect on the order of buffers. If you want to change the order of buffers in the fundamental buffer list, here is an easy way:

```
(defun reorder-buffer-list (new-list)
  (while new-list
    (bury-buffer (car new-list))
    (setq new-list (cdr new-list))))
```

With this method, you can specify any order for the list, but there is no danger of losing a buffer or adding something that is not a valid live buffer.

To change the order or value of a specific frame's buffer list, set that frame's `buffer-list` parameter with `modify-frame-parameters` (see Section 30.4.1 [Parameter Access], page 805).

`other-buffer` **&optional** *buffer visible-ok frame* [Function]

This function returns the first buffer in the buffer list other than *buffer*. Usually, this is the buffer appearing in the most recently selected window (in frame *frame* or else the selected frame, see Section 30.10 [Input Focus], page 827), aside from *buffer*. Buffers whose names start with a space are not considered at all.

If *buffer* is not supplied (or if it is not a live buffer), then `other-buffer` returns the first buffer in the selected frame's local buffer list. (If *frame* is non-`nil`, it returns the first buffer in *frame*'s local buffer list instead.)

If *frame* has a non-`nil` `buffer-predicate` parameter, then `other-buffer` uses that predicate to decide which buffers to consider. It calls the predicate once for each buffer, and if the value is `nil`, that buffer is ignored. See Section 30.4.3.5 [Buffer Parameters], page 814.

If *visible-ok* is `nil`, `other-buffer` avoids returning a buffer visible in any window on any visible frame, except as a last resort. If *visible-ok* is non-`nil`, then it does not matter whether a buffer is displayed somewhere or not.

If no suitable buffer exists, the buffer `*scratch*` is returned (and created, if necessary).

`last-buffer` **&optional** *buffer visible-ok frame* [Function]

This function returns the last buffer in *frame*'s buffer list other than *buffer*. If *frame* is omitted or `nil`, it uses the selected frame's buffer list.

The argument *visible-ok* is handled as with `other-buffer`, see above. If no suitable buffer can be found, the buffer `*scratch*` is returned.

`bury-buffer` **&optional** *buffer-or-name* [Command]

This command puts *buffer-or-name* at the end of the buffer list, without changing the order of any of the other buffers on the list. This buffer therefore becomes the

least desirable candidate for `other-buffer` to return. The argument can be either a buffer itself or the name of one.

This function operates on each frame's `buffer-list` parameter as well as the fundamental buffer list; therefore, the buffer that you bury will come last in the value of `(buffer-list` *frame*`)` and in the value of `(buffer-list)`. In addition, it also puts the buffer at the end of the list of buffers of the selected window (see Section 29.14 [Window History], page 748) provided it is shown in that window.

If *buffer-or-name* is `nil` or omitted, this means to bury the current buffer. In addition, if the current buffer is displayed in the selected window (see Section 29.3 [Selecting Windows], page 698), this makes sure that the window is either deleted or another buffer is shown in it. More precisely, if the selected window is dedicated (see Section 29.15 [Dedicated Windows], page 751) and there are other windows on its frame, the window is deleted. If it is the only window on its frame and that frame is not the only frame on its terminal, the frame is dismissed by calling the function specified by `frame-auto-hide-function` (see Section 29.16 [Quitting Windows], page 752). Otherwise, it calls `switch-to-prev-buffer` (see Section 29.14 [Window History], page 748) to show another buffer in that window. If *buffer-or-name* is displayed in some other window, it remains displayed there.

To replace a buffer in all the windows that display it, use `replace-buffer-in-windows`, See Section 29.11 [Buffers and Windows], page 722.

`unbury-buffer` [Command]

This command switches to the last buffer in the local buffer list of the selected frame. More precisely, it calls the function `switch-to-buffer` (see Section 29.12 [Switching Buffers], page 723), to display the buffer returned by `last-buffer` (see above), in the selected window.

`buffer-list-update-hook` [Variable]

This is a normal hook run whenever the buffer list changes. Functions (implicitly) running this hook are `get-buffer-create` (see Section 28.9 [Creating Buffers], page 686), `rename-buffer` (see Section 28.3 [Buffer Names], page 676), `kill-buffer` (see Section 28.10 [Killing Buffers], page 687), `bury-buffer` (see above), and `select-window` (see Section 29.3 [Selecting Windows], page 698). This hook is not run for internal or temporary buffers created by `get-buffer-create` or `generate-new-buffer` with a non-`nil` argument *inhibit-buffer-hooks*.

Functions run by this hook should avoid calling `select-window` with a `nil` *norecord* argument since this may lead to infinite recursion.

`buffer-match-p` *condition buffer-or-name* **&optional** *arg* [Function]

This function checks if a buffer designated by `buffer-or-name` satisfies the specified *condition*. Optional third argument *arg* is passed to the predicate function in *condition*. A valid *condition* can be one of the following:

- A string, interpreted as a regular expression. The buffer satisfies the condition if the regular expression matches the buffer name.
- A predicate function, which should return non-`nil` if the buffer matches. If the function expects one argument, it is called with *buffer-or-name* as the argument;

if it expects 2 arguments, the first argument is *buffer-or-name* and the second is *arg* (or `nil` if *arg* is omitted).

- A cons-cell (*oper* . *expr*) where *oper* is one of

  `(not` *cond*`)`
  : Satisfied if *cond* doesn't satisfy `buffer-match-p` with the same buffer and `arg`.

  `(or` *conds*`...)`
  : Satisfied if *any* condition in *conds* satisfies `buffer-match-p`, with the same buffer and `arg`.

  `(and` *conds*`...)`
  : Satisfied if *all* the conditions in *conds* satisfy `buffer-match-p`, with the same buffer and `arg`.

  `derived-mode`
  : Satisfied if the buffer's major mode derives from *expr*. Note that this condition might fail to report a match if `buffer-match-p` is invoked before the major mode of the buffer has been established.

  `major-mode`
  : Satisfied if the buffer's major mode is equal to *expr*. Prefer using `derived-mode` instead, when both can work. Note that this condition might fail to report a match if `buffer-match-p` is invoked before the major mode of the buffer has been established.

- t Satisfied by any buffer. A convenient alternative to `""` (empty string) or `(and)` (empty conjunction).

`match-buffers` *condition* **&optional** *buffer-list arg* [Function]
: This function returns a list of all buffers that satisfy the *condition*. If no buffers match, the function returns `nil`. The argument *condition* is as defined in `buffer-match-p` above. By default, all the buffers are considered, but this can be restricted via the optional argument `buffer-list`, which should be a list of buffers to consider. Optional third argument *arg* will be passed to *condition* in the same way as `buffer-match-p` does.

28.9 Creating Buffers

This section describes the two primitives for creating buffers. `get-buffer-create` creates a buffer if it finds no existing buffer with the specified name; `generate-new-buffer` always creates a new buffer and gives it a unique name.

Both functions accept an optional argument *inhibit-buffer-hooks*. If it is non-`nil`, the buffer they create does not run the hooks `kill-buffer-hook`, `kill-buffer-query-functions` (see Section 28.10 [Killing Buffers], page 687), and `buffer-list-update-hook` (see Section 28.8 [Buffer List], page 683). This avoids slowing down internal or temporary buffers that are never presented to users or passed on to other applications.

Other functions you can use to create buffers include `with-output-to-temp-buffer` (see Section 41.8 [Temporary Displays], page 1150) and `create-file-buffer` (see Section 26.1

[Visiting Files], page 610). Starting a subprocess can also create a buffer (see Chapter 40 [Processes], page 1081).

`get-buffer-create` *buffer-or-name* **&optional** *inhibit-buffer-hooks* [Function]
This function returns a buffer named *buffer-or-name*. The buffer returned does not become the current buffer—this function does not change which buffer is current.

buffer-or-name must be either a string or an existing buffer. If it is a string and a live buffer with that name already exists, `get-buffer-create` returns that buffer. If no such buffer exists, it creates a new buffer. If *buffer-or-name* is a buffer instead of a string, it is returned as given, even if it is dead.

```
(get-buffer-create "foo")
     ⇒ #<buffer foo>
```

The major mode for a newly created buffer is set to Fundamental mode. (The default value of the variable `major-mode` is handled at a higher level; see Section 24.2.2 [Auto Major Mode], page 532.) If the name begins with a space, the buffer initially disables undo information recording (see Section 33.9 [Undo], page 898).

`generate-new-buffer` *name* **&optional** *inhibit-buffer-hooks* [Function]
This function returns a newly created, empty buffer, but does not make it current. The name of the buffer is generated by passing *name* to the function `generate-new-buffer-name` (see Section 28.3 [Buffer Names], page 676). Thus, if there is no buffer named *name*, then that is the name of the new buffer; if that name is in use, a suffix of the form '`<n>`', where *n* is an integer, is appended to *name*.

An error is signaled if *name* is not a string.

```
(generate-new-buffer "bar")
     ⇒ #<buffer bar>
(generate-new-buffer "bar")
     ⇒ #<buffer bar<2>>
(generate-new-buffer "bar")
     ⇒ #<buffer bar<3>>
```

The major mode for the new buffer is set to Fundamental mode. The default value of the variable `major-mode` is handled at a higher level. See Section 24.2.2 [Auto Major Mode], page 532.

28.10 Killing Buffers

Killing a buffer makes its name unknown to Emacs and makes the memory space it occupied available for other use.

The buffer object for the buffer that has been killed remains in existence as long as anything refers to it, but it is specially marked so that you cannot make it current or display it. Killed buffers retain their identity, however; if you kill two distinct buffers, they remain distinct according to `eq` although both are dead.

If you kill a buffer that is current or displayed in a window, Emacs automatically selects or displays some other buffer instead. This means that killing a buffer can change the current buffer. Therefore, when you kill a buffer, you should also take the precautions

associated with changing the current buffer (unless you happen to know that the buffer being killed isn't current). See Section 28.2 [Current Buffer], page 673.

If you kill a buffer that is the base buffer of one or more indirect buffers, the indirect buffers are automatically killed as well.

The `buffer-name` of a buffer is `nil` if, and only if, the buffer is killed. A buffer that has not been killed is called a *live* buffer. To test whether a buffer is live or killed, use the function `buffer-live-p` (see below).

`kill-buffer` **&optional** *buffer-or-name* [Command]

This function kills the buffer *buffer-or-name*, freeing all its memory for other uses or to be returned to the operating system. If *buffer-or-name* is `nil` or omitted, it kills the current buffer.

Any processes that have this buffer as the `process-buffer` are sent the `SIGHUP` (hangup) signal, which normally causes them to terminate. See Section 40.8 [Signals to Processes], page 1100.

If the buffer is visiting a file and contains unsaved changes, `kill-buffer` asks the user to confirm before the buffer is killed. It does this even if not called interactively. To prevent the request for confirmation, clear the modified flag before calling `kill-buffer`. See Section 28.5 [Buffer Modification], page 679.

This function calls `replace-buffer-in-windows` for cleaning up all windows currently displaying the buffer to be killed.

Killing a buffer that is already dead has no effect.

This function returns `t` if it actually killed the buffer. It returns `nil` if the user refuses to confirm or if *buffer-or-name* was already dead.

```
(kill-buffer "foo.unchanged")
     ⇒ t
(kill-buffer "foo.changed")

---------- Buffer: Minibuffer ----------
Buffer foo.changed modified; kill anyway? (yes or no) yes
---------- Buffer: Minibuffer ----------

     ⇒ t
```

`kill-buffer-query-functions` [Variable]

Before confirming unsaved changes, `kill-buffer` calls the functions in the list `kill-buffer-query-functions`, in order of appearance, with no arguments. The buffer being killed is the current buffer when they are called. The idea of this feature is that these functions will ask for confirmation from the user. If any of them returns `nil`, `kill-buffer` spares the buffer's life.

This hook is not run for internal or temporary buffers created by `get-buffer-create` or `generate-new-buffer` with a non-`nil` argument *inhibit-buffer-hooks*.

`kill-buffer-hook` [Variable]

This is a normal hook run by `kill-buffer` after asking all the questions it is going to ask, just before actually killing the buffer. The buffer to be killed is current when the hook functions run. See Section 24.1 [Hooks], page 525. This variable is a permanent local, so its local binding is not cleared by changing major modes.

This hook is not run for internal or temporary buffers created by `get-buffer-create` or `generate-new-buffer` with a non-`nil` argument *inhibit-buffer-hooks*.

`buffer-offer-save` [User Option]
This variable, if non-`nil` in a particular buffer, tells `save-buffers-kill-emacs` to offer to save that buffer, just as it offers to save file-visiting buffers. If `save-some-buffers` is called with the second optional argument set to `t`, it will also offer to save the buffer. Lastly, if this variable is set to the symbol `always`, both `save-buffers-kill-emacs` and `save-some-buffers` will always offer to save. See [Definition of save-some-buffers], page 614. The variable `buffer-offer-save` automatically becomes buffer-local when set for any reason. See Section 12.11 [Buffer-Local Variables], page 207.

`buffer-save-without-query` [Variable]
This variable, if non-`nil` in a particular buffer, tells `save-buffers-kill-emacs` and `save-some-buffers` to save this buffer (if it's modified) without asking the user. The variable automatically becomes buffer-local when set for any reason.

`buffer-live-p` *object* [Function]
This function returns `t` if *object* is a live buffer (a buffer which has not been killed), `nil` otherwise.

28.11 Indirect Buffers

An *indirect buffer* shares the text of some other buffer, which is called the *base buffer* of the indirect buffer. In some ways it is the analogue, for buffers, of a symbolic link among files. The base buffer may not itself be an indirect buffer.

The text of the indirect buffer is always identical to the text of its base buffer; changes made by editing either one are visible immediately in the other. This includes the text properties as well as the characters themselves.

In all other respects, the indirect buffer and its base buffer are completely separate. They have different names, independent values of point, independent narrowing, independent markers and overlays (though inserting or deleting text in either buffer relocates the markers and overlays for both), independent major modes, and independent buffer-local variable bindings.

An indirect buffer cannot visit a file, but its base buffer can. If you try to save the indirect buffer, that actually saves the base buffer.

Killing an indirect buffer has no effect on its base buffer. Killing the base buffer effectively kills the indirect buffer in that it cannot ever again be the current buffer.

`make-indirect-buffer` *base-buffer name* **&optional** *clone inhibit-buffer-hooks* [Command]
This creates and returns an indirect buffer named *name* whose base buffer is *base-buffer*. The argument *base-buffer* may be a live buffer or the name (a string) of an existing buffer. If *name* is the name of an existing buffer, an error is signaled.

If *clone* is non-`nil`, then the indirect buffer originally shares the state of *base-buffer* such as major mode, minor modes, buffer local variables and so on. If *clone* is omitted or `nil` the indirect buffer's state is set to the default state for new buffers.

If *base-buffer* is an indirect buffer, its base buffer is used as the base for the new buffer. If, in addition, *clone* is non-`nil`, the initial state is copied from the actual base buffer, not from *base-buffer*.

See Section 28.9 [Creating Buffers], page 686, for the meaning of *inhibit-buffer-hooks*.

`clone-indirect-buffer` *newname display-flag* **&optional** *norecord* [Command]
This function creates and returns a new indirect buffer that shares the current buffer's base buffer and copies the rest of the current buffer's attributes. (If the current buffer is not indirect, it is used as the base buffer.)

If *display-flag* is non-`nil`, as it always is in interactive calls, that means to display the new buffer by calling `pop-to-buffer`. If *norecord* is non-`nil`, that means not to put the new buffer to the front of the buffer list.

`buffer-base-buffer` **&optional** *buffer* [Function]
This function returns the base buffer of *buffer*, which defaults to the current buffer. If *buffer* is not indirect, the value is `nil`. Otherwise, the value is another buffer, which is never an indirect buffer.

28.12 Swapping Text Between Two Buffers

Specialized modes sometimes need to let the user access from the same buffer several vastly different types of text. For example, you may need to display a summary of the buffer text, in addition to letting the user access the text itself.

This could be implemented with multiple buffers (kept in sync when the user edits the text), or with narrowing (see Section 31.4 [Narrowing], page 867). But these alternatives might sometimes become tedious or prohibitively expensive, especially if each type of text requires expensive buffer-global operations in order to provide correct display and editing commands.

Emacs provides another facility for such modes: you can quickly swap buffer text between two buffers with `buffer-swap-text`. This function is very fast because it doesn't move any text, it only changes the internal data structures of the buffer object to point to a different chunk of text. Using it, you can pretend that a group of two or more buffers are actually a single virtual buffer that holds the contents of all the individual buffers together.

`buffer-swap-text` *buffer* [Function]
This function swaps the text of the current buffer and that of its argument *buffer*. It signals an error if one of the two buffers is an indirect buffer (see Section 28.11 [Indirect Buffers], page 689) or is a base buffer of an indirect buffer.

All the buffer properties that are related to the buffer text are swapped as well: the positions of point and mark, all the markers, the overlays, the text properties, the undo list, the value of the `enable-multibyte-characters` flag (see Section 34.1 [Text Representations], page 966), etc.

Warning: If this function is called from within a `save-excursion` form, the current buffer will be set to *buffer* upon leaving the form, since the marker used by `save-excursion` to save the position and buffer will be swapped as well.

If you use `buffer-swap-text` on a file-visiting buffer, you should set up a hook to save the buffer's original text rather than what it was swapped with. `write-region-annotate-functions` works for this purpose. You should probably set `buffer-saved-size` to -2 in the buffer, so that changes in the text it is swapped with will not interfere with auto-saving.

28.13 The Buffer Gap

Emacs buffers are implemented using an invisible *gap* to make insertion and deletion faster. Insertion works by filling in part of the gap, and deletion adds to the gap. Of course, this means that the gap must first be moved to the locus of the insertion or deletion. Emacs moves the gap only when you try to insert or delete. This is why your first editing command in one part of a large buffer, after previously editing in another far-away part, sometimes involves a noticeable delay.

This mechanism works invisibly, and Lisp code should never be affected by the gap's current location, but these functions are available for getting information about the gap status.

`gap-position` [Function]
: This function returns the current gap position in the current buffer.

`gap-size` [Function]
: This function returns the current gap size of the current buffer.

29 Windows

This chapter describes the functions and variables related to Emacs windows. See Chapter 30 [Frames], page 787, for how windows are assigned an area of screen available for Emacs to use. See Chapter 41 [Display], page 1133, for information on how text is displayed in windows.

29.1 Basic Concepts of Emacs Windows

A *window* is an area of the screen that can be used to display a buffer (see Chapter 28 [Buffers], page 673). Windows are grouped into frames (see Chapter 30 [Frames], page 787). Each frame contains at least one window; the user can subdivide a frame into multiple, non-overlapping windows to view several buffers at once. Lisp programs can use multiple windows for a variety of purposes. In Rmail, for example, you can view a summary of message titles in one window, and the contents of the selected message in another window.

Emacs uses the term "window" with a different meaning than in graphical desktop environments and window systems, such as the X Window System. When Emacs is run on X, each graphical X window owned by the Emacs process corresponds to one Emacs frame. When Emacs is run on a text terminal, each Emacs frame fills the entire terminal screen. In either case, the frame may contain one or more Emacs windows. For disambiguation, we use the term *window-system window* when we mean the window-system window corresponding to an Emacs frame.

Unlike X windows, Emacs windows are *tiled*; they never overlap within the area of their frame. When a window is created, resized, or deleted, the change in window space is taken from or given to other windows on the same frame, so that the total area of the frame is unchanged.

In Emacs Lisp, windows are represented by a special Lisp object type (see Section 2.5.3 [Window Type], page 27).

`windowp` *object* [Function]
: This function returns `t` if *object* is a window (whether or not it displays a buffer). Otherwise, it returns `nil`.

A *live window* is one that is actually displaying a buffer in a frame.

`window-live-p` *object* [Function]
: This function returns `t` if *object* is a live window and `nil` otherwise. A live window is one that displays a buffer.

The windows in each frame are organized into a *window tree*. See Section 29.2 [Windows and Frames], page 694. The leaf nodes of each window tree are live windows—the ones actually displaying buffers. The internal nodes of the window tree are *internal windows*, which are not live.

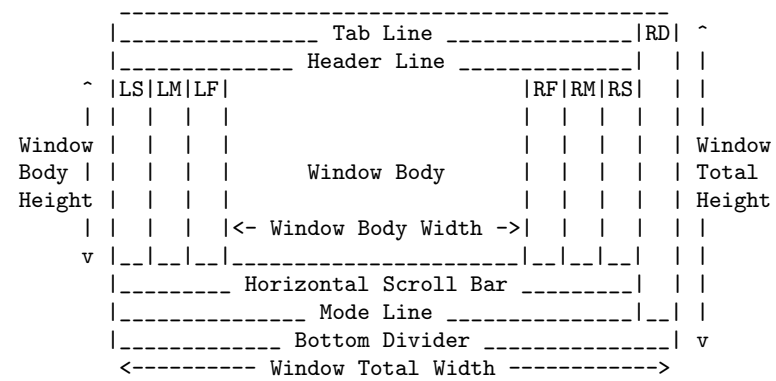
A *valid window* is one that is either live or internal. A valid window can be *deleted*, i.e., removed from its frame (see Section 29.8 [Deleting Windows], page 713); then it is no longer valid, but the Lisp object representing it might be still referenced from other Lisp objects. A deleted window may be made valid again by restoring a saved window configuration (see Section 29.26 [Window Configurations], page 777).

You can distinguish valid windows from deleted windows with `window-valid-p`.

`window-valid-p` *object* [Function]

This function returns `t` if *object* is a live window, or an internal window in a window tree. Otherwise, it returns `nil`, including for the case where *object* is a deleted window.

The following schematic shows the structure of a live window:



At the center of that window is the *body*, where the buffer text is displayed. The body can be surrounded by a series of optional areas which we will call *window decorations*. On the left and right, from innermost to outermost, these are the left and right fringes, denoted by LF and RF (see Section 41.13 [Fringes], page 1193); the left and right margins, denoted by LM and RM in the schematic (see Section 41.16.5 [Display Margins], page 1209); the left or right vertical scroll bar, only one of which is present at any time, denoted by LS and RS (see Section 41.14 [Scroll Bars], page 1199); and the right divider, denoted by RD (see Section 41.15 [Window Dividers], page 1202). Together these are the window's *left and right decorations*.

At the top of the window are the tab line and the header line (see Section 24.4.7 [Header Lines], page 560). The *text area* of the window includes the header line and the tab line, if they are present in the window. At the bottom of the window are the horizontal scroll bar (see Section 41.14 [Scroll Bars], page 1199); the mode line (see Section 24.4 [Mode Line Format], page 551); and the bottom divider (see Section 41.15 [Window Dividers], page 1202). Together these form the window's *top and bottom decorations*.

There are two special areas omitted in the schematic:

- When any of the fringes is missing, the display engine may use one character cell in its place for showing a continuation or truncation glyph provided a text line doesn't fit in a window.
- When both, the vertical scroll bar and the right divider are missing, the display engine usurps one pixel for drawing a vertical divider line between this window and the window on its right, provided such a window exists. On a text terminal, this divider always occupies an entire character cell.

In either case, the resulting artifact is considered part of the window's body although its screen space cannot be used for displaying buffer text.

Note also, that line numbers (and their surrounding whitespace) as displayed by `display-line-numbers-mode` (see Section "Display Custom" in *The GNU Emacs Manual*) do not count as decorations either; they are part of the window's body too.

Internal windows neither show any text nor do they have decorations. Hence, the concept of "body" does not make sense for them. In fact, most functions operating on the body of a window will yield an error when applied to an internal window.

By default, an Emacs frame exhibits one special live window that is used for displaying messages and accepting user input—the *minibuffer window* (see Section 21.11 [Minibuffer Windows], page 419). Since the minibuffer window is used for displaying text, it has a body but it does not have a tab or header line or any margins. Finally, a *tooltip window* which is used for displaying a tooltip in a tooltip frame (see Section 41.26 [Tooltips], page 1254) has a body too but no decorations at all.

29.2 Windows and Frames

Each window belongs to exactly one frame (see Chapter 30 [Frames], page 787). For all windows belonging to a specific frame, we sometimes also say that these windows are *owned* by that frame or simply that they are on that frame.

`window-frame` **&optional** *window* [Function]
: This function returns the specified *window*'s frame—the frame that *window* belongs to. If *window* is omitted or `nil`, it defaults to the selected window (see Section 29.3 [Selecting Windows], page 698).

`window-list` **&optional** *frame minibuffer window* [Function]
: This function returns a list of all live windows owned by the specified *frame*. If *frame* is omitted or `nil`, it defaults to the selected frame (see Section 30.10 [Input Focus], page 827).

  The optional argument *minibuffer* specifies whether to include the minibuffer window (see Section 21.11 [Minibuffer Windows], page 419) in that list. If *minibuffer* is `t`, the minibuffer window is included. If `nil` or omitted, the minibuffer window is included only if it is active. If *minibuffer* is neither `nil` nor `t`, the minibuffer window is never included.

  The optional argument *window*, if non-`nil`, must be a live window on the specified frame; then *window* will be the first element in the returned list. If *window* is omitted or `nil`, the window selected within *frame* (see Section 29.3 [Selecting Windows], page 698) is the first element.

Windows on the same frame are organized into a *window tree*, whose leaf nodes are the live windows. The internal nodes of a window tree are not live; they exist for the purpose of organizing the relationships between live windows. The root node of a window tree is called the *root window*. It is either a live window or an internal window. If it is a live window, then the frame has just one window besides the minibuffer window, or the frame is a minibuffer-only frame, see Section 30.3.1 [Frame Layout], page 793.

A minibuffer window (see Section 21.11 [Minibuffer Windows], page 419) that is not alone on its frame does not have a parent window, so it strictly speaking is not part of its frame's window tree. Nonetheless, it is a sibling window of the frame's root window, and thus can be reached from the root window via `window-next-sibling`, see below. Also, the function `window-tree` described at the end of this section lists the minibuffer window alongside the actual window tree.

`frame-root-window` **&optional** *frame-or-window* [Function]
: This function returns the root window for *frame-or-window*. The argument *frame-or-window* should be either a window or a frame; if omitted or `nil`, it defaults to the selected frame. If *frame-or-window* is a window, the return value is the root window of that window's frame.

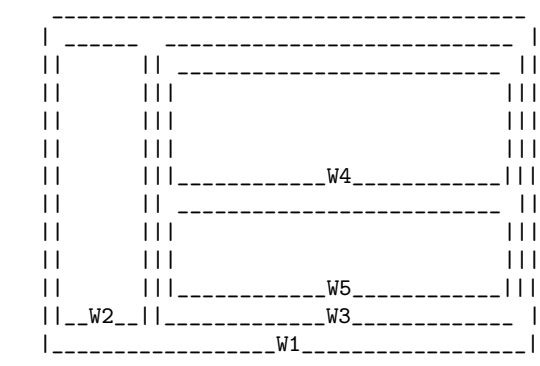
When a live window is split (see Section 29.7 [Splitting Windows], page 710), there are two live windows where previously there was one. One of these is represented by the same Lisp window object as the original window, and the other is represented by a newly-created Lisp window object. Both of these live windows become leaf nodes of the window tree, as *child windows* of a single internal window. If necessary, Emacs automatically creates this internal window, which is also called the *parent window*, and assigns it to the appropriate position in the window tree. The set of windows that share the same parent are called *siblings*.

`window-parent` **&optional** *window* [Function]
: This function returns the parent window of *window*. If *window* is omitted or `nil`, it defaults to the selected window. The return value is `nil` if *window* has no parent (i.e., it is a minibuffer window or the root window of its frame).

A parent window always has at least two child windows. If this number were to fall to one as a result of window deletion (see Section 29.8 [Deleting Windows], page 713), Emacs automatically deletes the parent window too, and its sole remaining child window takes its place in the window tree.

A child window can be either a live window, or an internal window (which in turn would have its own child windows). Therefore, each internal window can be thought of as occupying a certain rectangular *screen area*—the union of the areas occupied by the live windows that are ultimately descended from it.

For each internal window, the screen areas of the immediate children are arranged either vertically or horizontally (never both). If the child windows are arranged one above the other, they are said to form a *vertical combination*; if they are arranged side by side, they are said to form a *horizontal combination*. Consider the following example:



The root window of this frame is an internal window, *W1*. Its child windows form a horizontal combination, consisting of the live window *W2* and the internal window *W3*. The child windows of *W3* form a vertical combination, consisting of the live windows *W4* and *W5*. Hence, the live windows in this window tree are *W2*, *W4*, and *W5*.

The following functions can be used to retrieve a child window of an internal window, and the siblings of a child window. Their *window* argument always defaults to the selected window (see Section 29.3 [Selecting Windows], page 698).

`window-top-child` **&optional** *window* [Function]
: This function returns the topmost child window of *window*, if *window* is an internal window whose children form a vertical combination. For any other type of window, the return value is `nil`.

`window-left-child` **&optional** *window* [Function]
: This function returns the leftmost child window of *window*, if *window* is an internal window whose children form a horizontal combination. For any other type of window, the return value is `nil`.

`window-child` *window* [Function]
: This function returns the first child window of the internal window *window*—the topmost child window for a vertical combination, or the leftmost child window for a horizontal combination. If *window* is a live window, the return value is `nil`.

`window-combined-p` **&optional** *window horizontal* [Function]
: This function returns a non-`nil` value if and only if *window* is part of a vertical combination.

  If the optional argument *horizontal* is non-`nil`, this means to return non-`nil` if and only if *window* is part of a horizontal combination.

`window-next-sibling` **&optional** *window* [Function]
: This function returns the next sibling of the specified *window*. The return value is `nil` if *window* is the last child of its parent.

`window-prev-sibling` **&optional** *window* [Function]
: This function returns the previous sibling of the specified *window*. The return value is `nil` if *window* is the first child of its parent.

The functions `window-next-sibling` and `window-prev-sibling` should not be confused with the functions `next-window` and `previous-window`, which return the next and previous window in the cyclic ordering of windows (see Section 29.10 [Cyclic Window Ordering], page 719).

The following functions can be useful to locate a window within its frame.

`frame-first-window` **&optional** *frame-or-window* [Function]
: This function returns the live window at the upper left corner of the frame specified by *frame-or-window*. The argument *frame-or-window* must denote a window or a live frame and defaults to the selected frame. If *frame-or-window* specifies a window, this function returns the first window on that window's frame. Under the assumption that the frame from our canonical example is selected `(frame-first-window)` returns *W2*.

`window-at-side-p` **&optional** *window side* [Function]
: This function returns `t` if *window* is located at *side* of its containing frame. The argument *window* must be a valid window and defaults to the selected one. The

argument *side* can be any of the symbols `left`, `top`, `right` or `bottom`. The default value `nil` is handled like `bottom`.

Note that this function disregards the minibuffer window (see Section 21.11 [Minibuffer Windows], page 419). Hence, with *side* equal to `bottom` it may return `t` also when the minibuffer window appears right below *window*.

`window-in-direction` *direction* **&optional** *window ignore sign wrap minibuf* [Function]

This function returns the nearest live window in direction *direction* as seen from the position of `window-point` in window *window*. The argument *direction* must be one of `above`, `below`, `left` or `right`. The optional argument *window* must denote a live window and defaults to the selected one.

This function does not return a window whose `no-other-window` parameter is non-`nil` (see Section 29.27 [Window Parameters], page 779). If the nearest window's `no-other-window` parameter is non-`nil`, this function tries to find another window in the indicated direction whose `no-other-window` parameter is `nil`. If the optional argument *ignore* is non-`nil`, a window may be returned even if its `no-other-window` parameter is non-`nil`.

If the optional argument *sign* is a negative number, it means to use the right or bottom edge of *window* as reference position instead of `window-point`. If *sign* is a positive number, it means to use the left or top edge of *window* as reference position.

If the optional argument *wrap* is non-`nil`, this means to wrap *direction* around frame borders. For example, if *window* is at the top of the frame and *direction* is `above`, then this function usually returns the frame's minibuffer window if it's active and a window at the bottom of the frame otherwise.

If the optional argument *minibuf* is `t`, this function may return the minibuffer window even when it's not active. If the optional argument *minibuf* is `nil`, this means to return the minibuffer window if and only if it is currently active. If *minibuf* is neither `nil` nor `t`, this function never returns the minibuffer window. However, if *wrap* is non-`nil`, it always acts as if *minibuf* were `nil`.

If it doesn't find a suitable window, this function returns `nil`.

Don't use this function to check whether there is *no* window in *direction*. Calling `window-at-side-p` described above is a much more efficient way to do that.

The following function retrieves the entire window tree of a frame:

`window-tree` **&optional** *frame* [Function]

This function returns a list representing the window tree for frame *frame*. If *frame* is omitted or `nil`, it defaults to the selected frame.

The return value is a list of the form `(`*`root mini`*`)`, where *root* represents the window tree of the frame's root window, and *mini* is the frame's minibuffer window.

If the root window is live, *root* is that window itself. Otherwise, *root* is a list `(`*`dir edges w1 w2 ...`*`)` where *dir* is `nil` for a horizontal combination and `t` for a vertical combination, *edges* gives the size and position of the combination, and the remaining elements are the child windows. Each child window may again be a window object (for a live window) or a list with the same format as above (for an internal window).

The *edges* element is a list `(left top right bottom)`, similar to the value returned by `window-edges` (see Section 29.24 [Coordinates and Windows], page 772).

29.3 Selecting Windows

In each frame, at any time, exactly one Emacs window is designated as *selected within the frame*. For the selected frame, that window is called the *selected window*—the one in which most editing takes place, and in which the cursor for selected windows appears (see Section 30.4.3.9 [Cursor Parameters], page 819). Keyboard input that inserts or deletes text is also normally directed to this window. The selected window's buffer is usually also the current buffer, except when `set-buffer` has been used (see Section 28.2 [Current Buffer], page 673). As for non-selected frames, the window selected within the frame becomes the selected window if the frame is ever selected.

`selected-window` [Function]
This function returns the selected window (which is always a live window).

The following function explicitly selects a window and its frame.

`select-window` *window* **&optional** *norecord* [Function]
This function makes *window* the selected window and the window selected within its frame, and selects that frame. It also makes *window*'s buffer (see Section 29.11 [Buffers and Windows], page 722) current and sets that buffer's value of `point` to the value of `window-point` (see Section 29.19 [Window Point], page 761) in *window*. *window* must be a live window. The return value is *window*.

By default, this function also moves *window*'s buffer to the front of the buffer list (see Section 28.8 [Buffer List], page 683) and makes *window* the most recently selected window. If the optional argument *norecord* is non-`nil`, these additional actions are omitted.

In addition, this function by default also tells the display engine to update the display of *window* when its frame gets redisplayed the next time. If *norecord* is non-`nil`, such updates are usually not performed. If, however, *norecord* equals the special symbol `mark-for-redisplay`, the additional actions mentioned above are omitted but *window*'s display will be nevertheless updated.

Note that sometimes selecting a window is not enough to show it, or make its frame the top-most frame on display: you may also need to raise the frame or make sure input focus is directed to that frame. See Section 30.10 [Input Focus], page 827.

For historical reasons, Emacs does not run a separate hook whenever a window gets selected. Applications and internal routines often temporarily select a window to perform a few actions on it. They do that either to simplify coding—because many functions by default operate on the selected window when no *window* argument is specified—or because some functions did not (and still do not) take a window as argument and always operate(d) on the selected window instead. Running a hook every time a window gets selected for a short time and once more when the previously selected window gets restored is not useful.

However, when its *norecord* argument is `nil`, `select-window` updates the buffer list and thus indirectly runs the normal hook `buffer-list-update-hook` (see Section 28.8 [Buffer

List], page 683). Consequently, that hook provides one way to run a function whenever a window gets selected more "permanently".

Since `buffer-list-update-hook` is also run by functions that are not related to window management, it will usually make sense to save the value of the selected window somewhere and compare it with the value of `selected-window` while running that hook. Also, to avoid false positives when using `buffer-list-update-hook`, it is good practice that every `select-window` call supposed to select a window only temporarily passes a non-`nil` *norecord* argument. If possible, the macro `with-selected-window` (see below) should be used in such cases.

Emacs also runs the hook `window-selection-change-functions` whenever the redisplay routine detects that another window has been selected since last redisplay. See Section 29.28 [Window Hooks], page 782, for a detailed explanation. `window-state-change-functions` (described in the same section) is another abnormal hook run after a different window has been selected but is triggered by other window changes as well.

The sequence of calls to `select-window` with a non-`nil` *norecord* argument determines an ordering of windows by their selection or use time, see below. The function `get-lru-window`, for example, can then be used to retrieve the least recently selected window (see Section 29.10 [Cyclic Window Ordering], page 719).

`frame-selected-window` **&optional** *frame* [Function]
: This function returns the window on *frame* that is selected within that frame. *frame* should be a live frame; if omitted or `nil`, it defaults to the selected frame.

`set-frame-selected-window` *frame window* **&optional** *norecord* [Function]
: This function makes *window* the window selected within the frame *frame*. *frame* should be a live frame; if `nil`, it defaults to the selected frame. *window* must denote a live window.

  If *frame* is the selected frame, this makes *window* the selected window.

  If the optional argument *norecord* is non-`nil`, this function does not alter the ordering of the most recently selected windows, nor the buffer list.

The following macros are useful to temporarily select a window without affecting the ordering of recently selected windows or the buffer list.

`save-selected-window` *forms...* [Macro]
: This macro records the selected frame, as well as the selected window of each frame, executes *forms* in sequence, then restores the earlier selected frame and windows. It also saves and restores the current buffer. It returns the value of the last form in *forms*.

  This macro does not save or restore anything about the sizes, arrangement or contents of windows; therefore, if *forms* change them, the change persists. If the previously selected window of some frame is no longer live at the time of exit from *forms*, that frame's selected window is left alone. If the previously selected window is no longer live, then whatever window is selected at the end of *forms* remains selected. The current buffer is restored if and only if it is still live when exiting *forms*.

  This macro changes neither the ordering of recently selected windows nor the buffer list.

`with-selected-window` *window forms...* [Macro]
This macro selects *window*, executes *forms* in sequence, then restores the previously selected window and current buffer. The ordering of recently selected windows and the buffer list remain unchanged unless you deliberately change them within *forms*; for example, by calling `select-window` with argument *norecord* `nil`. Hence, this macro is the preferred way to temporarily work with *window* as the selected window without needlessly running `buffer-list-update-hook`.

Note that this macro temporarily puts the window management code in an unstable state. In particular, the most recently used window (see below) will not necessarily match the selected one. Hence, functions like `get-lru-window` and `get-mru-window` may return unexpected results when called from the body of this macro.

`with-selected-frame` *frame forms...* [Macro]
This macro executes *forms* with *frame* as the selected frame. The value returned is the value of the last form in *forms*. This macro saves and restores the selected frame, and changes the order of neither the recently selected windows nor the buffers in the buffer list.

`window-use-time` **&optional** *window* [Function]
This function returns the use time of window *window*. *window* must be a live window and defaults to the selected one.

The *use time* of a window is not really a time value, but an integer that does increase monotonically with each call of `select-window` with a `nil` *norecord* argument. The window with the lowest use time is usually called the least recently used window. The window with the highest use time is called the most recently used one (see Section 29.10 [Cyclic Window Ordering], page 719) and is usually the selected window unless `with-selected-window` has been used.

`window-bump-use-time` **&optional** *window* [Function]
This function marks *window* as being the second most recently used one (after the selected window). It does nothing if *window* is the selected window or the selected window does not have the highest use time among all windows which may happen within the scope of `with-selected-window`.

Sometimes several windows collectively and cooperatively display a buffer, for example, under the management of Follow Mode (see Section "Follow Mode" in `emacs`), where the windows together display a bigger portion of the buffer than one window could alone. It is often useful to consider such a *window group* as a single entity. Several functions such as `window-group-start` (see Section 29.20 [Window Start and End], page 762) allow you to do this by supplying, as an argument, one of the windows as a stand-in for the whole group.

`selected-window-group` [Function]
When the selected window is a member of a group of windows, this function returns a list of the windows in the group, ordered such that the first window in the list is displaying the earliest part of the buffer, and so on. Otherwise the function returns a list containing just the selected window.

The selected window is considered part of a group when the buffer local variable `selected-window-group-function` is set to a function. In this case,

`selected-window-group` calls it with no arguments and returns its result (which should be the list of windows in the group).

29.4 Window Sizes

Emacs provides miscellaneous functions for finding the height and width of a window. The return value of many of these functions can be specified either in units of pixels or in units of lines and columns. On a graphical display, the latter actually correspond to the height and width of a default character specified by the frame's default font as returned by `frame-char-height` and `frame-char-width` (see Section 30.3.2 [Frame Font], page 799). Thus, if a window is displaying text with a different font or size, the reported line height and column width for that window may differ from the actual number of text lines or columns displayed within it.

The *total height* of a window is the number of lines comprising its body and its top and bottom decorations (see Section 29.1 [Basic Windows], page 692).

`window-total-height` **&optional** *window round* [Function]

This function returns the total height, in lines, of the window *window*. If *window* is omitted or `nil`, it defaults to the selected window. If *window* is an internal window, the return value is the total height occupied by its descendant windows.

If a window's pixel height is not an integral multiple of its frame's default character height, the number of lines occupied by the window is rounded internally. This is done in a way such that, if the window is a parent window, the sum of the total heights of all its child windows internally equals the total height of their parent. This means that although two windows have the same pixel height, their internal total heights may differ by one line. This means also, that if window is vertically combined and has a next sibling, the topmost row of that sibling can be calculated as the sum of this window's topmost row and total height (see Section 29.24 [Coordinates and Windows], page 772)

If the optional argument *round* is `ceiling`, this function returns the smallest integer larger than *window*'s pixel height divided by the character height of its frame; if it is `floor`, it returns the largest integer smaller than said value; with any other *round* it returns the internal value of *windows*'s total height.

The *total width* of a window is the number of columns comprising its body and its left and right decorations (see Section 29.1 [Basic Windows], page 692).

`window-total-width` **&optional** *window round* [Function]

This function returns the total width, in columns, of the window *window*. If *window* is omitted or `nil`, it defaults to the selected window. If *window* is internal, the return value is the total width occupied by its descendant windows.

If a window's pixel width is not an integral multiple of its frame's character width, the number of columns occupied by the window is rounded internally. This is done in a way such that, if the window is a parent window, the sum of the total widths of all its children internally equals the total width of their parent. This means that although two windows have the same pixel width, their internal total widths may differ by one column. This means also, that if this window is horizontally combined

and has a next sibling, the leftmost column of that sibling can be calculated as the sum of this window's leftmost column and total width (see Section 29.24 [Coordinates and Windows], page 772). The optional argument *round* behaves as it does for `window-total-height`.

`window-total-size` **&optional** *window horizontal round* [Function]
This function returns either the total height in lines or the total width in columns of the window *window*. If *horizontal* is omitted or `nil`, this is equivalent to calling `window-total-height` for *window*; otherwise it is equivalent to calling `window-total-width` for *window*. The optional argument *round* behaves as it does for `window-total-height`.

The following two functions can be used to return the total size of a window in units of pixels.

`window-pixel-height` **&optional** *window* [Function]
This function returns the total height of window *window* in pixels. *window* must be a valid window and defaults to the selected one.

The return value includes the heights of *window*'s top and bottom decorations. If *window* is an internal window, its pixel height is the pixel height of the screen areas spanned by its children.

`window-pixel-width` **&optional** *window* [Function]
This function returns the width of window *window* in pixels. *window* must be a valid window and defaults to the selected one.

The return value includes the widths of *window*'s left and right decorations. If *window* is an internal window, its pixel width is the width of the screen areas spanned by its children.

The following functions can be used to determine whether a given window has any adjacent windows.

`window-full-height-p` **&optional** *window* [Function]
This function returns non-`nil` if *window* has no other window above or below it in its frame. More precisely, this means that the total height of *window* equals the total height of the root window on that frame. The minibuffer window does not count in this regard. If *window* is omitted or `nil`, it defaults to the selected window.

`window-full-width-p` **&optional** *window* [Function]
This function returns non-`nil` if *window* has no other window to the left or right in its frame, i.e., its total width equals that of the root window on that frame. If *window* is omitted or `nil`, it defaults to the selected window.

The *body height* of a window is the height of its body, which does not include any of its top or bottom decorations (see Section 29.1 [Basic Windows], page 692).

`window-body-height` **&optional** *window pixelwise* [Function]
This function returns the height, in lines, of the body of window *window*. If *window* is omitted or `nil`, it defaults to the selected window; otherwise it must be a live window.

The optional argument *pixelwise* defines the units to use for the height. If `nil`, return the body height of *window* in characters, rounded down to the nearest integer, if necessary. This means that if a line at the bottom of the text area is only partially visible, that line is not counted. It also means that the height of a window's body can never exceed its total height as returned by `window-total-height`.

If *pixelwise* is `remap` and the default face is remapped (see Section 41.12.5 [Face Remapping], page 1179), use the remapped face to determine the character height. For any other non-`nil` value, return the height in pixels.

The *body width* of a window is the width of its body and of the text area, which does not include any of its left or right decorations (see Section 29.1 [Basic Windows], page 692).

Note that when one or both fringes are removed (by setting their width to zero), the display engine reserves two character cells, one on each side of the window, for displaying the continuation and truncation glyphs, which leaves 2 columns less for text display. (The function `window-max-chars-per-line`, described below, takes this peculiarity into account.)

`window-body-width` **&optional** *window pixelwise* [Function]

This function returns the width, in columns, of the body of window *window*. If *window* is omitted or `nil`, it defaults to the selected window; otherwise it must be a live window.

The optional argument *pixelwise* defines the units to use for the width. If `nil`, return the body width of *window* in characters, rounded down to the nearest integer, if necessary. This means that if a column on the right of the text area is only partially visible, that column is not counted. It also means that the width of a window's body can never exceed its total width as returned by `window-total-width`.

If *pixelwise* is `remap` and the default face is remapped (see Section 41.12.5 [Face Remapping], page 1179), use the remapped face to determine the character width. For any other non-`nil` value, return the width in pixels.

`window-body-size` **&optional** *window horizontal pixelwise* [Function]

This function returns the body height or body width of *window*. If *horizontal* is omitted or `nil`, it is equivalent to calling `window-body-height` for *window*; otherwise it is equivalent to calling `window-body-width`. In either case, the optional argument *pixelwise* is passed to the function called.

The pixel heights of a window's mode, tab and header line can be retrieved with the functions given below. Their return value is usually accurate unless the window has not been displayed before: In that case, the return value is based on an estimate of the font used for the window's frame.

`window-mode-line-height` **&optional** *window* [Function]

This function returns the height in pixels of *window*'s mode line. *window* must be a live window and defaults to the selected one. If *window* has no mode line, the return value is zero.

`window-tab-line-height` **&optional** *window* [Function]
: This function returns the height in pixels of *window*'s tab line. *window* must be a live window and defaults to the selected one. If *window* has no tab line, the return value is zero.

`window-header-line-height` **&optional** *window* [Function]
: This function returns the height in pixels of *window*'s header line. *window* must be a live window and defaults to the selected one. If *window* has no header line, the return value is zero.

Functions for retrieving the height and/or width of window dividers (see Section 41.15 [Window Dividers], page 1202), fringes (see Section 41.13 [Fringes], page 1193), scroll bars (see Section 41.14 [Scroll Bars], page 1199), and display margins (see Section 41.16.5 [Display Margins], page 1209) are described in the corresponding sections.

If your Lisp program needs to make layout decisions, you will find the following function useful:

`window-max-chars-per-line` **&optional** *window face* [Function]
: This function returns the number of characters displayed in the specified face *face* in the specified window *window* (which must be a live window). If *face* was remapped (see Section 41.12.5 [Face Remapping], page 1179), the information is returned for the remapped face. If omitted or `nil`, *face* defaults to the default face, and *window* defaults to the selected window.

  Unlike `window-body-width`, this function accounts for the actual size of *face*'s font, instead of working in units of the canonical character width of *window*'s frame (see Section 30.3.2 [Frame Font], page 799). It also accounts for space used by the continuation glyph, if *window* lacks one or both of its fringes.

Commands that change the size of windows (see Section 29.5 [Resizing Windows], page 705), or split them (see Section 29.7 [Splitting Windows], page 710), obey the variables `window-min-height` and `window-min-width`, which specify the smallest allowable window height and width. They also obey the variable `window-size-fixed`, with which a window can be *fixed* in size (see Section 29.6 [Preserving Window Sizes], page 709).

`window-min-height` [User Option]
: This option specifies the minimum total height, in lines, of any window. Its value has to accommodate at least one text line and any top or bottom decorations.

`window-min-width` [User Option]
: This option specifies the minimum total width, in columns, of any window. Its value has to accommodate at least two text columns and any left or right decorations.

The following function tells how small a specific window can get taking into account the sizes of its areas and the values of `window-min-height`, `window-min-width` and `window-size-fixed` (see Section 29.6 [Preserving Window Sizes], page 709).

`window-min-size` **&optional** *window horizontal ignore pixelwise* [Function]
: This function returns the minimum size of *window*. *window* must be a valid window and defaults to the selected one. The optional argument *horizontal* non-`nil` means to

return the minimum number of columns of *window*; otherwise return the minimum number of *window*'s lines.

The return value makes sure that all components of *window* remain fully visible if *window*'s size were actually set to it. With *horizontal* `nil` it includes any top or bottom decorations. With *horizontal* non-`nil` it includes any left or right decorations of *window*.

The optional argument *ignore*, if non-`nil`, means ignore restrictions imposed by fixed size windows, `window-min-height` or `window-min-width` settings. If *ignore* equals `safe`, live windows may get as small as `window-safe-min-height` lines and `window-safe-min-width` columns. If *ignore* is a window, ignore restrictions for that window only. Any other non-`nil` value means ignore all of the above restrictions for all windows.

The optional argument *pixelwise* non-`nil` means to return the minimum size of *window* counted in pixels.

29.5 Resizing Windows

This section describes functions for resizing a window without changing the size of its frame. Because live windows do not overlap, these functions are meaningful only on frames that contain two or more windows: resizing a window also changes the size of at least one other window. If there is just one window on a frame, its size cannot be changed except by resizing the frame (see Section 30.3.4 [Frame Size], page 801).

Except where noted, these functions also accept internal windows as arguments. Resizing an internal window causes its child windows to be resized to fit the same space.

`window-resizable` *window delta* **&optional** *horizontal ignore pixelwise* [Function]

This function returns *delta* if the size of *window* can be changed vertically by *delta* lines. If the optional argument *horizontal* is non-`nil`, it instead returns *delta* if *window* can be resized horizontally by *delta* columns. It does not actually change the window size.

If *window* is `nil`, it defaults to the selected window.

A positive value of *delta* means to check whether the window can be enlarged by that number of lines or columns; a negative value of *delta* means to check whether the window can be shrunk by that many lines or columns. If *delta* is non-zero, a return value of 0 means that the window cannot be resized.

Normally, the variables `window-min-height` and `window-min-width` specify the smallest allowable window size (see Section 29.4 [Window Sizes], page 701). However, if the optional argument *ignore* is non-`nil`, this function ignores `window-min-height` and `window-min-width`, as well as `window-size-fixed`. Instead, it considers the minimum height of a window as the sum of its top and bottom decorations plus the text of one line; and its minimum width as the sum of its left and right decorations plus text that takes two columns.

If the optional argument *pixelwise* is non-`nil`, *delta* is interpreted as pixels.

`window-resize` *window delta* **&optional** *horizontal ignore pixelwise* [Function]
This function resizes *window* by *delta* increments. If *horizontal* is `nil`, it changes the height by *delta* lines; otherwise, it changes the width by *delta* columns. A positive *delta* means to enlarge the window, and a negative *delta* means to shrink it.

If *window* is `nil`, it defaults to the selected window. If the window cannot be resized as demanded, an error is signaled.

The optional argument *ignore* has the same meaning as for the function `window-resizable` above.

If the optional argument *pixelwise* is non-`nil`, *delta* will be interpreted as pixels.

The choice of which window edges this function alters depends on the values of the option `window-combination-resize` and the combination limits of the involved windows; in some cases, it may alter both edges. See Section 29.9 [Recombining Windows], page 714. To resize by moving only the bottom or right edge of a window, use the function `adjust-window-trailing-edge`.

`adjust-window-trailing-edge` *window delta* **&optional** *horizontal pixelwise* [Function]
This function moves *window*'s bottom edge by *delta* lines. If optional argument *horizontal* is non-`nil`, it instead moves the right edge by *delta* columns. If *window* is `nil`, it defaults to the selected window.

If the optional argument *pixelwise* is non-`nil`, *delta* is interpreted as pixels.

A positive *delta* moves the edge downwards or to the right; a negative *delta* moves it upwards or to the left. If the edge cannot be moved as far as specified by *delta*, this function moves it as far as possible but does not signal an error.

This function tries to resize windows adjacent to the edge that is moved. If this is not possible for some reason (e.g., if that adjacent window is fixed-size), it may resize other windows.

`window-resize-pixelwise` [User Option]
If the value of this option is non-`nil`, Emacs resizes windows in units of pixels. This currently affects functions like `split-window` (see Section 29.7 [Splitting Windows], page 710), `maximize-window`, `minimize-window`, `fit-window-to-buffer`, `fit-frame-to-buffer` and `shrink-window-if-larger-than-buffer` (all listed below).

Note that when a frame's pixel size is not a multiple of its character size, at least one window may get resized pixelwise even if this option is `nil`. The default value is `nil`.

The following commands resize windows in more specific ways. When called interactively, they act on the selected window.

`fit-window-to-buffer` **&optional** *window max-height min-height max-width min-width preserve-size* [Command]
This command adjusts the height or width of *window* to fit the text in it. It returns non-`nil` if it was able to resize *window*, and `nil` otherwise. If *window* is omitted or `nil`, it defaults to the selected window. Otherwise, it should be a live window.

If *window* is part of a vertical combination, this function adjusts *window*'s height. The new height is calculated from the actual height of the accessible portion of its

buffer. The optional argument *max-height*, if non-`nil`, specifies the maximum total height that this function can give *window*. The optional argument *min-height*, if non-`nil`, specifies the minimum total height that it can give, which overrides the variable `window-min-height`. Both *max-height* and *min-height* are specified in lines and include any top or bottom decorations of *window*.

If *window* is part of a horizontal combination and the value of the option `fit-window-to-buffer-horizontally` (see below) is non-`nil`, this function adjusts *window*'s width. The new width of *window* is calculated from the maximum length of its buffer's lines that follow the current start position of *window*. The optional argument *max-width* specifies a maximum width and defaults to the width of *window*'s frame. The optional argument *min-width* specifies a minimum width and defaults to `window-min-width`. Both *max-width* and *min-width* are specified in columns and include any left or right decorations of *window*.

The optional argument *preserve-size*, if non-`nil`, will install a parameter to preserve the size of *window* during future resize operations (see Section 29.6 [Preserving Window Sizes], page 709).

If the option `fit-frame-to-buffer` (see below) is non-`nil`, this function will try to resize the frame of *window* to fit its contents by calling `fit-frame-to-buffer` (see below).

`fit-window-to-buffer-horizontally` [User Option]

If this is non-`nil`, `fit-window-to-buffer` can resize windows horizontally. If this is `nil` (the default) `fit-window-to-buffer` never resizes windows horizontally. If this is `only`, it can resize windows horizontally only. Any other value means `fit-window-to-buffer` can resize windows in both dimensions.

`fit-frame-to-buffer` [User Option]

If this option is non-`nil`, `fit-window-to-buffer` can fit a frame to its buffer. A frame is fit if and only if its root window is a live window and this option is non-`nil`. If this is `horizontally`, frames are fit horizontally only. If this is `vertically`, frames are fit vertically only. Any other non-`nil` value means frames can be resized in both dimensions.

If you have a frame that displays only one window, you can fit that frame to its buffer using the command `fit-frame-to-buffer`.

`fit-frame-to-buffer` **&optional** *frame max-height min-height max-width min-width only* [Command]

This command adjusts the size of *frame* to display the contents of its buffer exactly. *frame* can be any live frame and defaults to the selected one. Fitting is done only if *frame*'s root window is live.

The arguments *max-height*, *min-height*, *max-width* and *min-width*, if non-`nil`, specify bounds on the new body size of *frame*'s root window. A non-`nil` value specified by any of these arguments overrides the corresponding value specified by the option `fit-frame-to-buffer-sizes` described below.

If the optional argument *only* is `vertically`, this function may resize the frame vertically only. If *only* is `horizontally`, it may resize the frame horizontally only.

The behavior of `fit-frame-to-buffer` can be controlled with the help of the two options listed next.

`fit-frame-to-buffer-margins` [User Option]
: This option can be used to specify margins around frames to be fit by `fit-frame-to-buffer`. Such margins can be useful to avoid, for example, that the resized frame overlaps the taskbar or parts of its parent frame.

  It specifies the numbers of pixels to be left free on the left, above, the right, and below a frame that shall be fit. The default specifies `nil` for each which means to use no margins. The value specified here can be overridden for a specific frame by that frame's `fit-frame-to-buffer-margins` parameter, if present.

`fit-frame-to-buffer-sizes` [User Option]
: This option specifies size boundaries for `fit-frame-to-buffer`. It specifies the maximum and minimum lines and maximum and minimum columns of the root window's body of any frame that shall be fit to its buffer. Any value this option specifies will be overridden by the corresponding argument of `fit-frame-to-buffer`, if non-`nil`.

`shrink-window-if-larger-than-buffer` **&optional** *window* [Command]
: This command attempts to reduce *window*'s height as much as possible while still showing its full buffer, but no less than `window-min-height` lines. The return value is non-`nil` if the window was resized, and `nil` otherwise. If *window* is omitted or `nil`, it defaults to the selected window. Otherwise, it should be a live window.

  This command does nothing if the window is already too short to display all of its buffer, or if any of the buffer is scrolled off-screen, or if the window is the only live window in its frame.

  This command calls `fit-window-to-buffer` (see above) to do its work.

`balance-windows` **&optional** *window-or-frame* [Command]
: This function balances windows in a way that gives more space to full-width and/or full-height windows. If *window-or-frame* specifies a frame, it balances all windows on that frame. If *window-or-frame* specifies a window, it balances only that window and its siblings (see Section 29.2 [Windows and Frames], page 694).

`balance-windows-area` [Command]
: This function attempts to give all windows on the selected frame approximately the same share of the screen area. Full-width or full-height windows are not given more space than other windows.

`maximize-window` **&optional** *window* [Command]
: This function attempts to make *window* as large as possible, in both dimensions, without resizing its frame or deleting other windows. If *window* is omitted or `nil`, it defaults to the selected window.

`minimize-window` **&optional** *window* [Command]
: This function attempts to make *window* as small as possible, in both dimensions, without deleting it or resizing its frame. If *window* is omitted or `nil`, it defaults to the selected window.

29.6 Preserving Window Sizes

A window can get resized explicitly by using one of the functions from the preceding section or implicitly, for example, when resizing an adjacent window, when splitting or deleting a window (see Section 29.7 [Splitting Windows], page 710, see Section 29.8 [Deleting Windows], page 713) or when resizing the window's frame (see Section 30.3.4 [Frame Size], page 801).

It is possible to avoid implicit resizing of a specific window when there are one or more other resizable windows on the same frame. For this purpose, Emacs must be advised to *preserve* the size of that window. There are two basic ways to do that.

`window-size-fixed` [Variable]

If this buffer-local variable is non-`nil`, the size of any window displaying the buffer cannot normally be changed. Deleting a window or changing the frame's size may still change the window's size, if there is no choice.

If the value is `height`, then only the window's height is fixed; if the value is `width`, then only the window's width is fixed. Any other non-`nil` value fixes both the width and the height.

If this variable is `nil`, this does not necessarily mean that any window showing the buffer can be resized in the desired direction. To determine that, use the function `window-resizable`. See Section 29.5 [Resizing Windows], page 705.

Often `window-size-fixed` is overly aggressive because it inhibits any attempt to explicitly resize or split an affected window as well. This may even happen after the window has been resized implicitly, for example, when deleting an adjacent window or resizing the window's frame. The following function tries hard to never disallow resizing such a window explicitly:

`window-preserve-size` **&optional** *window horizontal preserve* [Function]

This function (un-)marks the height of window *window* as preserved for future resize operations. *window* must be a live window and defaults to the selected one. If the optional argument *horizontal* is non-`nil`, it (un-)marks the width of *window* as preserved.

If the optional argument *preserve* is `t`, this means to preserve the current height/width of *window*'s body. The height/width of *window* will change only if Emacs has no better choice. Resizing a window whose height/width is preserved by this function never throws an error.

If *preserve* is `nil`, this means to stop preserving the height/width of *window*, lifting any respective restraint induced by a previous call of this function for *window*. Calling `enlarge-window`, `shrink-window` or `fit-window-to-buffer` with *window* as argument may also remove the respective restraint.

`window-preserve-size` is currently invoked by the following functions:

`fit-window-to-buffer`

If the optional argument *preserve-size* of that function (see Section 29.5 [Resizing Windows], page 705) is non-`nil`, the size established by that function is preserved.

`display-buffer`
: If the *alist* argument of that function (see Section 29.13.1 [Choosing Window], page 726) contains a `preserve-size` entry, the size of the window produced by that function is preserved.

`window-preserve-size` installs a window parameter (see Section 29.27 [Window Parameters], page 779) called `window-preserved-size` which is consulted by the window resizing functions. This parameter will not prevent resizing the window when the window shows another buffer than the one when `window-preserve-size` was invoked or if its size has changed since then.

The following function can be used to check whether the height of a particular window is preserved:

`window-preserved-size` **&optional** *window horizontal* [Function]
: This function returns the preserved height of window *window* in pixels. *window* must be a live window and defaults to the selected one. If the optional argument *horizontal* is non-`nil`, it returns the preserved width of *window*. It returns `nil` if the size of *window* is not preserved.

29.7 Splitting Windows

This section describes functions for creating a new window by *splitting* an existing one. Note that some windows are special in the sense that these functions may fail to split them as described here. Examples of such windows are side windows (see Section 29.17 [Side Windows], page 754) and atomic windows (see Section 29.18 [Atomic Windows], page 759).

`split-window` **&optional** *window size side pixelwise* [Function]
: This function creates a new live window next to the window *window*. If *window* is omitted or `nil`, it defaults to the selected window. That window is split, and reduced in size. The space is taken up by the new window, which is returned.

  The optional second argument *size* determines the sizes of *window* and/or the new window. If it is omitted or `nil`, both windows are given equal sizes; if there is an odd line, it is allocated to the new window. If *size* is a positive number, *window* is given *size* lines (or columns, depending on the value of *side*). If *size* is a negative number, the new window is given $-$*size* lines (or columns).

  If *size* is `nil`, this function obeys the variables `window-min-height` and `window-min-width` (see Section 29.4 [Window Sizes], page 701). Thus, it signals an error if splitting would result in making a window smaller than those variables specify. However, a non-`nil` value for *size* causes those variables to be ignored; in that case, the smallest allowable window is considered to be one that has space for a text that is one line tall and/or two columns wide.

  Hence, if *size* is specified, it's the caller's responsibility to check whether the emanating windows are large enough to encompass all of their decorations like a mode line or a scroll bar. The function `window-min-size` (see Section 29.4 [Window Sizes], page 701) can be used to determine the minimum requirements of *window* in this regard. Since the new window usually inherits areas like the mode line or the scroll bar from *window*, that function is also a good guess for the minimum size of the new

window. The caller should specify a smaller size only if it correspondingly removes an inherited area before the next redisplay.

The optional third argument *side* determines the position of the new window relative to *window*. If it is `nil` or `below`, the new window is placed below *window*. If it is `above`, the new window is placed above *window*. In both these cases, *size* specifies a total window height, in lines.

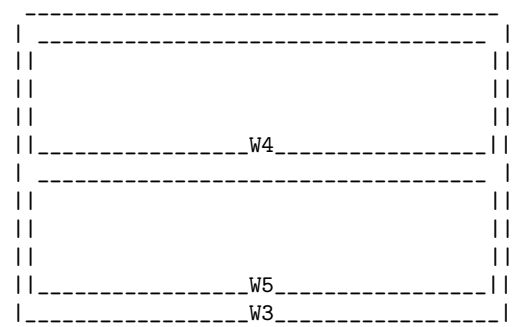
If *side* is `t` or `right`, the new window is placed on the right of *window*. If *side* is `left`, the new window is placed on the left of *window*. In both these cases, *size* specifies a total window width, in columns.

The optional fourth argument *pixelwise*, if non-`nil`, means to interpret *size* in units of pixels, instead of lines and columns.

If *window* is a live window, the new window inherits various properties from it, including margins and scroll bars. If *window* is an internal window, the new window inherits the properties of the window selected within *window*'s frame.

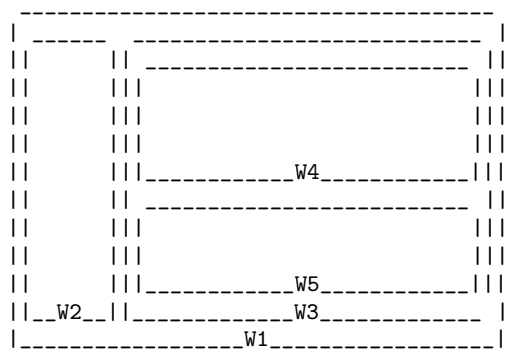
The behavior of this function may be altered by the window parameters of *window*, so long as the variable `ignore-window-parameters` is `nil`. If the value of the `split-window` window parameter is `t`, this function ignores all other window parameters. Otherwise, if the value of the `split-window` window parameter is a function, that function is called with the arguments *window*, *size*, and *side*, in lieu of the usual action of `split-window`. Otherwise, this function obeys the `window-atom` or `window-side` window parameter, if any. See Section 29.27 [Window Parameters], page 779.

As an example, here is a sequence of `split-window` calls that yields the window configuration discussed in Section 29.2 [Windows and Frames], page 694. This example demonstrates splitting a live window as well as splitting an internal window. We begin with a frame containing a single window (a live root window), which we denote by *W4*. Calling `(split-window W4)` yields this window configuration:



The `split-window` call has created a new live window, denoted by *W5*. It has also created a new internal window, denoted by *W3*, which becomes the root window and the parent of both *W4* and *W5*.

Next, we call `(split-window W3 nil 'left)`, passing the internal window *W3* as the argument. The result:



A new live window *W2* is created, to the left of the internal window *W3*. A new internal window *W1* is created, becoming the new root window.

For interactive use, Emacs provides two commands which always split the selected window. These call `split-window` internally.

`split-window-right` **&optional** *size window-to-split* [Command]
: This function splits the window *window-to-split* into two side-by-side windows, putting *window-to-split* on the left. *window-to-split* defaults to the selected window. If *size* is positive, the left window gets *size* columns; if *size* is negative, the right window gets $-$*size* columns.

`split-window-below` **&optional** *size window-to-split* [Command]
: This function splits the window *window-to-split* into two windows, one above the other, leaving the upper window selected. *window-to-split* defaults to the selected window. If *size* is positive, the upper window gets *size* lines; if *size* is negative, the lower window gets $-$*size* lines.

`split-root-window-below` **&optional** *size* [Command]
: This function splits the whole frame in two. The current window configuration is retained on the top, and a new window is created below, taking up the whole width of the frame. *size* is treated as by `split-window-below`.

`split-root-window-right` **&optional** *size* [Command]
: This function splits the whole frame in two. The current window configuration is retained on the left, and a new window is created on the right, taking up the whole height of the frame. *size* is treated as by `split-window-right`.

`split-window-keep-point` [User Option]
: If the value of this variable is non-`nil` (the default), `split-window-below` behaves as described above.

  If it is `nil`, `split-window-below` adjusts point in each of the two windows to minimize redisplay. (This is useful on slow terminals.) It selects whichever window contains the screen line that point was previously on. Note that this only affects `split-window-below`, not the lower-level `split-window` function.

29.8 Deleting Windows

Deleting a window removes it from the frame's window tree. If the window is a live window, it disappears from the screen. If the window is an internal window, its child windows are deleted too.

Even after a window is deleted, it continues to exist as a Lisp object, until there are no more references to it. Window deletion can be reversed, by restoring a saved window configuration (see Section 29.26 [Window Configurations], page 777).

`delete-window` **&optional** *window* [Command]

This function removes *window* from display and returns `nil`. If *window* is omitted or `nil`, it defaults to the selected window.

If deleting the window would leave no more windows in the window tree (e.g., if it is the only live window in the frame) or all remaining windows on *window*'s frame are side windows (see Section 29.17 [Side Windows], page 754), an error is signaled. If *window* is part of an atomic window (see Section 29.18 [Atomic Windows], page 759), this function tries to delete the root of that atomic window instead.

By default, the space taken up by *window* is given to one of its adjacent sibling windows, if any. However, if the variable `window-combination-resize` is non-`nil`, the space is proportionally distributed among any remaining windows in the same window combination. See Section 29.9 [Recombining Windows], page 714.

The behavior of this function may be altered by the window parameters of *window*, so long as the variable `ignore-window-parameters` is `nil`. If the value of the `delete-window` window parameter is `t`, this function ignores all other window parameters. Otherwise, if the value of the `delete-window` window parameter is a function, that function is called with the argument *window*, in lieu of the usual action of `delete-window`. See Section 29.27 [Window Parameters], page 779.

When `delete-window` deletes the selected window of its frame, it has to make another window the new selected window of that frame. The following option allows configuring which window is chosen.

`delete-window-choose-selected` [User Option]

This option allows specifying which window should become a frame's selected window after `delete-window` has deleted the previously selected one. Possible choices are

- `mru` (the default) choose the most recently used window on that frame.
- `pos` choose the window comprising the frame coordinates of point of the previously selected window on that frame.
- `nil` choose the first window (the window returned by `frame-first-window`) on that frame.

A window with a non-`nil` `no-other-window` parameter is chosen only if all other windows on that frame have that parameter set to a non-`nil` value too.

`delete-other-windows` **&optional** *window* [Command]

This function makes *window* fill its frame, deleting other windows as necessary. If *window* is omitted or `nil`, it defaults to the selected window. An error is signaled if *window* is a side window (see Section 29.17 [Side Windows], page 754). If *window*

is part of an atomic window (see Section 29.18 [Atomic Windows], page 759), this function tries to make the root of that atomic window fill its frame. The return value is `nil`.

The behavior of this function may be altered by the window parameters of *window*, so long as the variable `ignore-window-parameters` is `nil`. If the value of the `delete-other-windows` window parameter is `t`, this function ignores all other window parameters. Otherwise, if the value of the `delete-other-windows` window parameter is a function, that function is called with the argument *window*, in lieu of the usual action of `delete-other-windows`. See Section 29.27 [Window Parameters], page 779.

Also, if `ignore-window-parameters` is `nil`, this function does not delete any window whose `no-delete-other-windows` parameter is non-`nil`.

`delete-windows-on` **&optional** *buffer-or-name frame* [Command]
This function deletes all windows showing *buffer-or-name*, by calling `delete-window` on those windows. *buffer-or-name* should be a buffer, or the name of a buffer; if omitted or `nil`, it defaults to the current buffer. If there are no windows showing the specified buffer, this function does nothing. If the specified buffer is a minibuffer, an error is signaled.

If there is a dedicated window showing the buffer, and that window is the only one on its frame, this function also deletes that frame if it is not the only frame on the terminal.

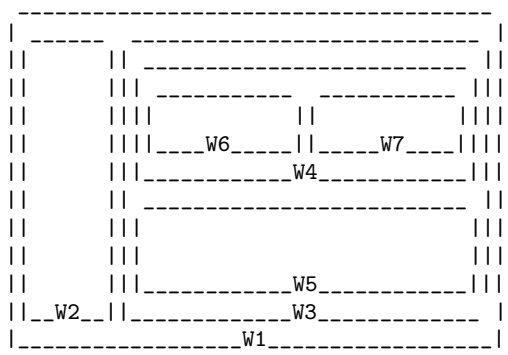
The optional argument *frame* specifies which frames to operate on:

- `nil` means operate on all frames.
- `t` means operate on the selected frame.
- `visible` means operate on all visible frames.
- `0` means operate on all visible or iconified frames.
- A frame means operate on that frame.

Note that this argument does not have the same meaning as in other functions which scan all live windows (see Section 29.10 [Cyclic Window Ordering], page 719). Specifically, the meanings of `t` and `nil` here are the opposite of what they are in those other functions.

29.9 Recombining Windows

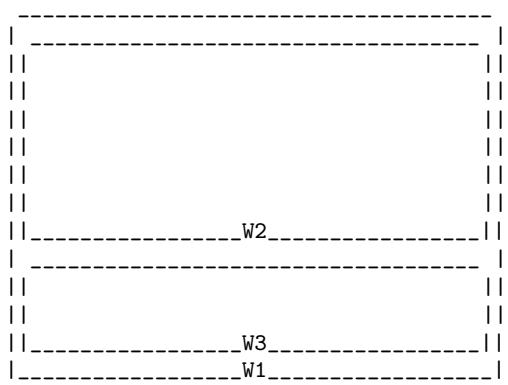
When deleting the last sibling of a window *W*, its parent window is deleted too, with *W* replacing it in the window tree. This means that *W* must be recombined with its parent's siblings to form a new window combination (see Section 29.2 [Windows and Frames], page 694). In some occasions, deleting a live window may even entail the deletion of two internal windows.



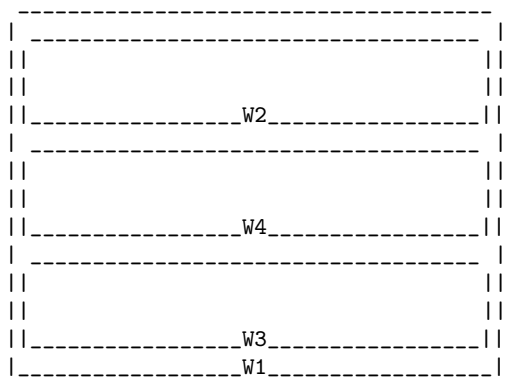
Deleting *W5* in this configuration normally causes the deletion of *W3* and *W4*. The remaining live windows *W2*, *W6* and *W7* are recombined to form a new horizontal combination with parent *W1*.

Sometimes, however, it makes sense to not delete a parent window like *W4*. In particular, a parent window should not be removed when it was used to preserve a combination embedded in a combination of the same type. Such embeddings make sense to assure that when you split a window and subsequently delete the new window, Emacs reestablishes the layout of the associated frame as it existed before the splitting.

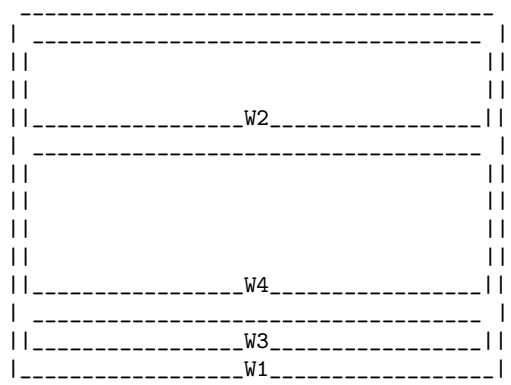
Consider a scenario starting with two live windows *W2* and *W3* and their parent *W1*.



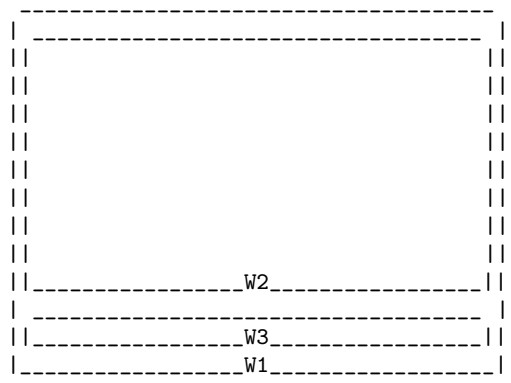
Split *W2* to make a new window *W4* as follows.



Now, when enlarging a window vertically, Emacs tries to obtain the corresponding space from its lower sibling, provided such a window exists. In our scenario, enlarging *W4* will steal space from *W3*.



Deleting *W4* will now give its entire space to *W2*, including the space earlier stolen from *W3*.



This can be counterintuitive, in particular if *W4* were used for displaying a buffer only temporarily (see Section 41.8 [Temporary Displays], page 1150), and you want to continue working with the initial layout.

The behavior can be fixed by making a new parent window when splitting *W2*. The variable described next allows that to be done.

`window-combination-limit` [User Option]

This variable controls whether splitting a window shall make a new parent window. The following values are recognized:

`nil` This means that the new live window is allowed to share the existing parent window, if one exists, provided the split occurs in the same direction as the existing window combination (otherwise, a new internal window is created anyway).

`window-size`
This means that `display-buffer` makes a new parent window when it splits a window and is passed a `window-height` or `window-width` entry in

the *alist* argument (see Section 29.13.2 [Buffer Display Action Functions], page 728). Otherwise, window splitting behaves as for a value of `nil`.

`temp-buffer-resize`
: In this case `with-temp-buffer-window` makes a new parent window when it splits a window and `temp-buffer-resize-mode` is enabled (see Section 41.8 [Temporary Displays], page 1150). Otherwise, window splitting behaves as for `nil`.

`temp-buffer`
: In this case `with-temp-buffer-window` always makes a new parent window when it splits an existing window (see Section 41.8 [Temporary Displays], page 1150). Otherwise, window splitting behaves as for `nil`.

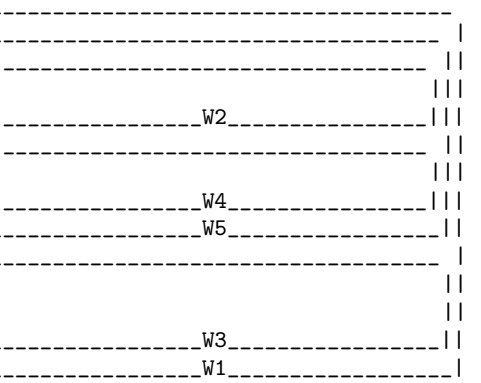
`display-buffer`
: This means that when `display-buffer` (see Section 29.13.1 [Choosing Window], page 726) splits a window it always makes a new parent window. Otherwise, window splitting behaves as for `nil`.

`t`
: This means that splitting a window always creates a new parent window. Thus, if the value of this variable is at all times `t`, then at all times every window tree is a binary tree (a tree where each window except the root window has exactly one sibling).

The default is `window-size`. Other values are reserved for future use.

If, as a consequence of this variable's setting, `split-window` makes a new parent window, it also calls `set-window-combination-limit` (see below) on the newly-created internal window. This affects how the window tree is rearranged when the child windows are deleted (see below).

If `window-combination-limit` is `t`, splitting *W2* in the initial configuration of our scenario would have produced this:



A new internal window *W5* has been created; its children are *W2* and the new live window *W4*. Now, *W2* is the only sibling of *W4*, so enlarging *W4* will try to shrink *W2*, leaving *W3* unaffected. Observe that *W5* represents a vertical combination of two windows embedded in the vertical combination *W1*.

`set-window-combination-limit` *window limit* [Function]

This function sets the *combination limit* of the window *window* to *limit*. This value can be retrieved via the function `window-combination-limit`. See below for its effects; note that it is only meaningful for internal windows. The `split-window` function automatically calls this function, passing it `t` as *limit*, provided the value of the variable `window-combination-limit` is `t` when it is called.

`window-combination-limit` *window* [Function]

This function returns the combination limit for *window*.

The combination limit is meaningful only for an internal window. If it is `nil`, then Emacs is allowed to automatically delete *window*, in response to a window deletion, in order to group the child windows of *window* with its sibling windows to form a new window combination. If the combination limit is `t`, the child windows of *window* are never automatically recombined with its siblings.

If, in the configuration shown at the beginning of this section, the combination limit of *W4* (the parent window of *W6* and *W7*) is `t`, deleting *W5* will not implicitly delete *W4* too.

Alternatively, the problems sketched above can be avoided by always resizing all windows in the same combination whenever one of its windows is split or deleted. This also permits splitting windows that would be otherwise too small for such an operation.

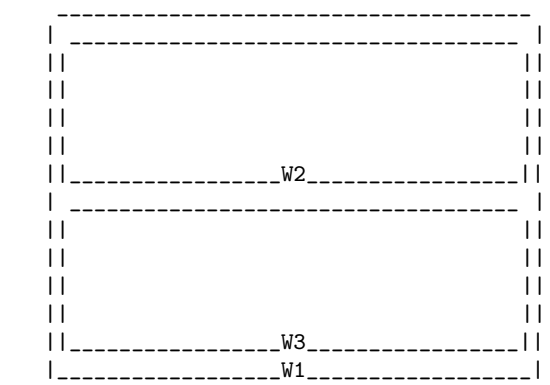
`window-combination-resize` [User Option]

If this variable is `nil`, `split-window` can only split a window (denoted by *window*) if *window*'s screen area is large enough to accommodate both itself and the new window.

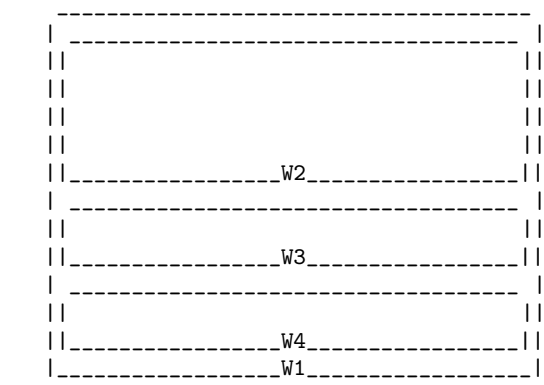
If this variable is `t`, `split-window` tries to resize all windows that are part of the same combination as *window*, in order to accommodate the new window. In particular, this may allow `split-window` to succeed even if *window* is a fixed-size window or too small to ordinarily split. Furthermore, subsequently resizing or deleting *window* may resize all other windows in its combination.

The default is `nil`. Other values are reserved for future use. A specific split operation may ignore the value of this variable if it is affected by a non-`nil` value of `window-combination-limit`.

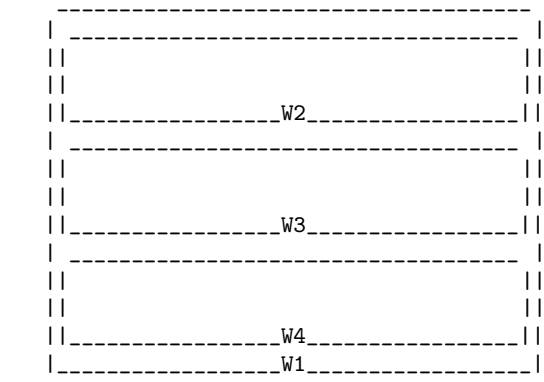
To illustrate the effect of `window-combination-resize`, consider the following frame layout.



If `window-combination-resize` is `nil`, splitting window *W3* leaves the size of *W2* unchanged:



If `window-combination-resize` is `t`, splitting *W3* instead leaves all three live windows with approximately the same height:



Deleting any of the live windows *W2*, *W3* or *W4* will distribute its space proportionally among the two remaining live windows.

29.10 Cyclic Ordering of Windows

When you use the command `C-x o` (`other-window`) to select some other window, it moves through live windows in a specific order. For any given configuration of windows, this order never varies. It is called the *cyclic ordering of windows*.

The ordering is determined by a depth-first traversal of each frame's window tree, retrieving the live windows which are the leaf nodes of the tree (see Section 29.2 [Windows and Frames], page 694). If the minibuffer is active, the minibuffer window is included too. The ordering is cyclic, so the last window in the sequence is followed by the first one.

`next-window` **&optional** *window minibuf all-frames* [Function]

This function returns a live window, the one following *window* in the cyclic ordering of windows. *window* should be a live window; if omitted or `nil`, it defaults to the selected window.

The optional argument *minibuf* specifies whether minibuffer windows should be included in the cyclic ordering. Normally, when *minibuf* is `nil`, a minibuffer window

is included only if it is currently active; this matches the behavior of `C-x o`. (Note that a minibuffer window is active as long as its minibuffer is in use; see Chapter 21 [Minibuffers], page 386).

If *minibuf* is `t`, the cyclic ordering includes all minibuffer windows. If *minibuf* is neither `t` nor `nil`, minibuffer windows are not included even if they are active.

The optional argument *all-frames* specifies which frames to consider:

- `nil` means to consider windows on *window*'s frame. If the minibuffer window is considered (as specified by the *minibuf* argument), then frames that share the minibuffer window are considered too.
- `t` means to consider windows on all existing frames.
- `visible` means to consider windows on all visible frames.
- 0 means to consider windows on all visible or iconified frames.
- A frame means to consider windows on that specific frame.
- Anything else means to consider windows on *window*'s frame, and no others.

If more than one frame is considered, the cyclic ordering is obtained by appending the orderings for those frames, in the same order as the list of all live frames (see Section 30.8 [Finding All Frames], page 825).

`previous-window` **&optional** *window minibuf all-frames* [Function]
This function returns a live window, the one preceding *window* in the cyclic ordering of windows. The other arguments are handled like in `next-window`.

`other-window` *count* **&optional** *all-frames* [Command]
This function selects a live window, one *count* places from the selected window in the cyclic ordering of windows. If *count* is a positive number, it skips *count* windows forwards; if *count* is negative, it skips −*count* windows backwards; if *count* is zero, that simply re-selects the selected window. When called interactively, *count* is the numeric prefix argument.

The optional argument *all-frames* has the same meaning as in `next-window`, like a `nil` *minibuf* argument to `next-window`.

This function does not select a window that has a non-`nil` `no-other-window` window parameter (see Section 29.27 [Window Parameters], page 779), provided that `ignore-window-parameters` is `nil`.

If the `other-window` parameter of the selected window is a function, and `ignore-window-parameters` is `nil`, that function will be called with the arguments *count* and *all-frames* instead of the normal operation of this function.

`walk-windows` *fun* **&optional** *minibuf all-frames* [Function]
This function calls the function *fun* once for each live window, with the window as the argument.

It follows the cyclic ordering of windows. The optional arguments *minibuf* and *all-frames* specify the set of windows included; these have the same arguments as in `next-window`. If *all-frames* specifies a frame, the first window walked is the first window on that frame (the one returned by `frame-first-window`), not necessarily the selected window.

If *fun* changes the window configuration by splitting or deleting windows, that does not alter the set of windows walked, which is determined prior to calling *fun* for the first time.

`one-window-p` **&optional** *no-mini all-frames* [Function]
This function returns `t` if the selected window is the only live window, and `nil` otherwise.

If the minibuffer window is active, it is normally considered (so that this function returns `nil`). However, if the optional argument *no-mini* is non-`nil`, the minibuffer window is ignored even if active. The optional argument *all-frames* has the same meaning as for `next-window`.

The following functions return a window which satisfies some criterion, without selecting it:

`get-lru-window` **&optional** *all-frames dedicated not-selected no-other* [Function]
This function returns a live window which is heuristically the least recently used one. The *least recently used window* is the least recently selected one—the window whose use time is less than the use time of all other live windows (see Section 29.3 [Selecting Windows], page 698). The optional argument *all-frames* has the same meaning as in `next-window`.

If any full-width windows are present, only those windows are considered. A minibuffer window is never a candidate. A dedicated window (see Section 29.15 [Dedicated Windows], page 751) is never a candidate unless the optional argument *dedicated* is non-`nil`. The selected window is never returned, unless it is the only candidate. However, if the optional argument *not-selected* is non-`nil`, this function returns `nil` in that case. The optional argument *no-other*, if non-`nil`, means to never return a window whose `no-other-window` parameter is non-`nil`.

`get-mru-window` **&optional** *all-frames dedicated not-selected no-other* [Function]
This function is like `get-lru-window`, but it returns the most recently used window instead. The *most recently used window* is the most recently selected one—the window whose use time exceeds the use time of all other live windows (see Section 29.3 [Selecting Windows], page 698). The meaning of the arguments is the same as for `get-lru-window`.

Since in practice the most recently used window is always the selected one, it usually makes sense to call this function with a non-`nil` *not-selected* argument only.

`get-largest-window` **&optional** *all-frames dedicated not-selected no-other* [Function]
This function returns the window with the largest area (height times width). If there are two candidate windows of the same size, it prefers the one that comes first in the cyclic ordering of windows, starting from the selected window. The meaning of the arguments is the same as for `get-lru-window`.

`get-window-with-predicate` *predicate* **&optional** *minibuf all-frames default* [Function]
This function calls the function *predicate* for each of the windows in the cyclic order of windows in turn, passing it the window as an argument. If the predicate returns

non-`nil` for any window, this function stops and returns that window. If no such window is found, the return value is *default* (which defaults to `nil`).

The optional arguments *minibuf* and *all-frames* specify the windows to search, and have the same meanings as in `next-window`.

29.11 Buffers and Windows

This section describes low-level functions for examining and setting the contents of windows. See Section 29.12 [Switching Buffers], page 723, for higher-level functions for displaying a specific buffer in a window.

`window-buffer` **&optional** *window* [Function]
This function returns the buffer that *window* is displaying. If *window* is omitted or `nil` it defaults to the selected window. If *window* is an internal window, this function returns `nil`.

`set-window-buffer` *window buffer-or-name* **&optional** *keep-margins* [Function]
This function makes *window* display *buffer-or-name*. *window* should be a live window; if `nil`, it defaults to the selected window. *buffer-or-name* should be a buffer, or the name of an existing buffer. This function does not change which window is selected, nor does it directly change which buffer is current (see Section 28.2 [Current Buffer], page 673). Its return value is `nil`.

If *window* is *strongly dedicated* to a buffer and *buffer-or-name* does not specify that buffer, this function signals an error. See Section 29.15 [Dedicated Windows], page 751.

By default, this function resets *window*'s position, display margins, fringe widths, and scroll bar settings, based on the local variables in the specified buffer. However, if the optional argument *keep-margins* is non-`nil`, it leaves *window*'s display margins, fringes and scroll bar settings alone.

When writing an application, you should normally use `display-buffer` (see Section 29.13.1 [Choosing Window], page 726) or the higher-level functions described in Section 29.12 [Switching Buffers], page 723, instead of calling `set-window-buffer` directly.

This runs `window-scroll-functions`, followed by `window-configuration-change-hook`. See Section 29.28 [Window Hooks], page 782.

`buffer-display-count` [Variable]
This buffer-local variable records the number of times a buffer has been displayed in a window. It is incremented each time `set-window-buffer` is called for the buffer.

`buffer-display-time` [Variable]
This buffer-local variable records the time at which a buffer was last displayed in a window. The value is `nil` if the buffer has never been displayed. It is updated each time `set-window-buffer` is called for the buffer, with the value returned by `current-time` (see Section 42.5 [Time of Day], page 1275).

`get-buffer-window` **&optional** *buffer-or-name all-frames* [Function]
This function returns the first window displaying *buffer-or-name* in the cyclic ordering of windows, starting from the selected window (see Section 29.10 [Cyclic Window Ordering], page 719). If no such window exists, the return value is `nil`.

buffer-or-name should be a buffer or the name of a buffer; if omitted or `nil`, it defaults to the current buffer. The optional argument *all-frames* specifies which windows to consider:

- `t` means consider windows on all existing frames.
- `visible` means consider windows on all visible frames.
- 0 means consider windows on all visible or iconified frames.
- A frame means consider windows on that frame only.
- Any other value means consider windows on the selected frame.

Note that these meanings differ slightly from those of the *all-frames* argument to `next-window` (see Section 29.10 [Cyclic Window Ordering], page 719). This function may be changed in a future version of Emacs to eliminate this discrepancy.

`get-buffer-window-list` **&optional** *buffer-or-name minibuf all-frames* [Function]
This function returns a list of all windows currently displaying *buffer-or-name*. *buffer-or-name* should be a buffer or the name of an existing buffer. If omitted or `nil`, it defaults to the current buffer. If the currently selected window displays *buffer-or-name*, it will be the first in the list returned by this function.

The arguments *minibuf* and *all-frames* have the same meanings as in the function `next-window` (see Section 29.10 [Cyclic Window Ordering], page 719). Note that the *all-frames* argument does *not* behave exactly like in `get-buffer-window`.

`replace-buffer-in-windows` **&optional** *buffer-or-name* [Command]
This command replaces *buffer-or-name* with some other buffer, in all windows displaying it. *buffer-or-name* should be a buffer, or the name of an existing buffer; if omitted or `nil`, it defaults to the current buffer.

The replacement buffer in each window is chosen via `switch-to-prev-buffer` (see Section 29.14 [Window History], page 748). With the exception of side windows (see Section 29.17 [Side Windows], page 754), any dedicated window displaying *buffer-or-name* is deleted if possible (see Section 29.15 [Dedicated Windows], page 751). If such a window is the only window on its frame and there are other frames on the same terminal, the frame is deleted as well. If the dedicated window is the only window on the only frame on its terminal, the buffer is replaced anyway.

29.12 Switching to a Buffer in a Window

This section describes high-level functions for switching to a specified buffer in some window. In general, "switching to a buffer" means to (1) show the buffer in some window, (2) make that window the selected window (and its frame the selected frame), and (3) make the buffer the current buffer.

Do *not* use these functions to make a buffer temporarily current just so a Lisp program can access or modify it. They have side-effects, such as changing window histories (see

Section 29.14 [Window History], page 748), which will surprise the user if used that way. If you want to make a buffer current to modify it in Lisp, use `with-current-buffer`, `save-current-buffer`, or `set-buffer`. See Section 28.2 [Current Buffer], page 673.

`switch-to-buffer` *buffer-or-name* **&optional** *norecord* *force-same-window* [Command]

This command attempts to display *buffer-or-name* in the selected window and make it the current buffer. It is often used interactively (as the binding of `C-x b`), as well as in Lisp programs. The return value is the buffer switched to.

If *buffer-or-name* is `nil`, it defaults to the buffer returned by `other-buffer` (see Section 28.8 [Buffer List], page 683). If *buffer-or-name* is a string that is not the name of any existing buffer, this function creates a new buffer with that name; the new buffer's major mode is determined by the variable `major-mode` (see Section 24.2 [Major Modes], page 527).

Normally, the specified buffer is put at the front of the buffer list—both the global buffer list and the selected frame's buffer list (see Section 28.8 [Buffer List], page 683). However, this is not done if the optional argument *norecord* is non-`nil`.

Sometimes, the selected window may not be suitable for displaying the buffer. This happens if the selected window is a minibuffer window, or if the selected window is strongly dedicated to its buffer (see Section 29.15 [Dedicated Windows], page 751). In such cases, the command normally tries to display the buffer in some other window, by invoking `pop-to-buffer` (see below).

If the optional argument *force-same-window* is non-`nil` and the selected window is not suitable for displaying the buffer, this function always signals an error when called non-interactively. In interactive use, if the selected window is a minibuffer window, this function will try to use some other window instead. If the selected window is strongly dedicated to its buffer, the option `switch-to-buffer-in-dedicated-window` described next can be used to proceed.

`switch-to-buffer-in-dedicated-window` [User Option]

This option, if non-`nil`, allows `switch-to-buffer` to proceed when called interactively and the selected window is strongly dedicated to its buffer.

The following values are respected:

`nil`	Disallows switching and signals an error as in non-interactive use.
`prompt`	Prompts the user whether to allow switching.
`pop`	Invokes `pop-to-buffer` to proceed.
`t`	Marks the selected window as non-dedicated and proceeds.

This option does not affect non-interactive calls of `switch-to-buffer`.

By default, `switch-to-buffer` tries to preserve `window-point`. This behavior can be tuned using the following option.

`switch-to-buffer-preserve-window-point` [User Option]

If this variable is `nil`, `switch-to-buffer` displays the buffer specified by *buffer-or-name* at the position of that buffer's `point`. If this variable is `already-displayed`, it

tries to display the buffer at its previous position in the selected window, provided the buffer is currently displayed in some other window on any visible or iconified frame. If this variable is `t`, `switch-to-buffer` unconditionally tries to display the buffer at its previous position in the selected window.

This variable is ignored if the buffer is already displayed in the selected window or never appeared in it before, or if `switch-to-buffer` calls `pop-to-buffer` to display the buffer.

`switch-to-buffer-obey-display-actions` [User Option]
If this variable is non-`nil`, `switch-to-buffer` respects display actions specified by `display-buffer-overriding-action`, `display-buffer-alist` and other display related variables.

The next two commands are similar to `switch-to-buffer`, except for the described features.

`switch-to-buffer-other-window` *buffer-or-name* **&optional** *norecord* [Command]
This function displays the buffer specified by *buffer-or-name* in some window other than the selected window. It uses the function `pop-to-buffer` internally (see below).

If the selected window already displays the specified buffer, it continues to do so, but another window is nonetheless found to display it as well.

The *buffer-or-name* and *norecord* arguments have the same meanings as in `switch-to-buffer`.

`switch-to-buffer-other-frame` *buffer-or-name* **&optional** *norecord* [Command]
This function displays the buffer specified by *buffer-or-name* in a new frame. It uses the function `pop-to-buffer` internally (see below).

If the specified buffer is already displayed in another window, in any frame on the current terminal, this switches to that window instead of creating a new frame. However, the selected window is never used for this.

The *buffer-or-name* and *norecord* arguments have the same meanings as in `switch-to-buffer`.

The above commands use the function `pop-to-buffer`, which flexibly displays a buffer in some window and selects that window for editing. In turn, `pop-to-buffer` uses `display-buffer` for displaying the buffer. Hence, all the variables affecting `display-buffer` will affect it as well. See Section 29.13.1 [Choosing Window], page 726, for the documentation of `display-buffer`.

`pop-to-buffer` *buffer-or-name* **&optional** *action norecord* [Command]
This function makes *buffer-or-name* the current buffer and displays it in some window, preferably not the window currently selected. It then selects the displaying window. If that window is on a different graphical frame, that frame is given input focus if possible (see Section 30.10 [Input Focus], page 827).

If *buffer-or-name* is `nil`, it defaults to the buffer returned by `other-buffer` (see Section 28.8 [Buffer List], page 683). If *buffer-or-name* is a string that is not the

name of any existing buffer, this function creates a new buffer with that name; the new buffer's major mode is determined by the variable `major-mode` (see Section 24.2 [Major Modes], page 527). In any case, that buffer is made current and returned, even when no suitable window was found to display it.

If *action* is non-`nil`, it should be a display action to pass to `display-buffer` (see Section 29.13.1 [Choosing Window], page 726). Alternatively, a non-`nil`, non-list value means to pop to a window other than the selected one—even if the buffer is already displayed in the selected window.

Like `switch-to-buffer`, this function updates the buffer list unless *norecord* is non-`nil`.

29.13 Displaying a Buffer in a Suitable Window

This section describes lower-level functions Emacs uses to find or create a window for displaying a specified buffer. The common workhorse of these functions is `display-buffer` which eventually handles all incoming requests for buffer display (see Section 29.13.1 [Choosing Window], page 726).

`display-buffer` delegates the task of finding a suitable window to so-called action functions (see Section 29.13.2 [Buffer Display Action Functions], page 728). First, `display-buffer` compiles a so-called action alist—a special association list that action functions can use to fine-tune their behavior. Then it passes that alist on to each action function it calls (see Section 29.13.3 [Buffer Display Action Alists], page 733).

The behavior of `display-buffer` is highly customizable. To understand how customizations are used in practice, you may wish to study examples illustrating the order of precedence which `display-buffer` uses to call action functions (see Section 29.13.5 [Precedence of Action Functions], page 741). To avoid conflicts between Lisp programs calling `display-buffer` and user customizations of its behavior, it may make sense to follow a number of guidelines which are sketched in the final part of this section (see Section 29.13.6 [The Zen of Buffer Display], page 745).

29.13.1 Choosing a Window for Displaying a Buffer

The command `display-buffer` flexibly chooses a window for display, and displays a specified buffer in that window. It can be called interactively, via the key binding `C-x 4 C-o`. It is also used as a subroutine by many functions and commands, including `switch-to-buffer` and `pop-to-buffer` (see Section 29.12 [Switching Buffers], page 723).

This command performs several complex steps to find a window to display in. These steps are described by means of *display actions*, which have the form `(functions . alist)`. Here, *functions* is either a single function or a list of functions, referred to as "action functions" (see Section 29.13.2 [Buffer Display Action Functions], page 728); and *alist* is an association list, referred to as "action alist" (see Section 29.13.3 [Buffer Display Action Alists], page 733). See Section 29.13.6 [The Zen of Buffer Display], page 745, for samples of display actions.

An action function accepts two arguments: the buffer to display and an action alist. It attempts to display the buffer in some window, picking or creating a window according to its own criteria. If successful, it returns the window; otherwise, it returns `nil`.

`display-buffer` works by combining display actions from several sources, and calling the action functions in turn, until one of them manages to display the buffer and returns a non-`nil` value.

`display-buffer` *buffer-or-name* **&optional** *action frame* [Command]

This command makes *buffer-or-name* appear in some window, without selecting the window or making the buffer current. The argument *buffer-or-name* must be a buffer or the name of an existing buffer. The return value is the window chosen to display the buffer, or `nil` if no suitable window was found.

The optional argument *action*, if non-`nil`, should normally be a display action (described above). `display-buffer` builds a list of action functions and an action alist, by consolidating display actions from the following sources (in order of their precedence, from highest to lowest):

- The variable `display-buffer-overriding-action`.
- The user option `display-buffer-alist`.
- The *action* argument.
- The user option `display-buffer-base-action`.
- The constant `display-buffer-fallback-action`.

In practice this means that `display-buffer` builds a list of all action functions specified by these display actions. The first element of this list is the first action function specified by `display-buffer-overriding-action`, if any. Its last element is `display-buffer-pop-up-frame`—the last action function specified by `display-buffer-fallback-action`. Duplicates are not removed from this list—hence one and the same action function may be called multiple times during one call of `display-buffer`.

`display-buffer` calls the action functions specified by this list in turn, passing the buffer as the first argument and the combined action alist as the second argument, until one of the functions returns non-`nil`. See Section 29.13.5 [Precedence of Action Functions], page 741, for examples how display actions specified by different sources are processed by `display-buffer`.

Note that the second argument is always the list of *all* action alist entries specified by the sources named above. Hence, the first element of that list is the first action alist entry specified by `display-buffer-overriding-action`, if any. Its last element is the last alist entry of `display-buffer-base-action`, if any (the action alist of `display-buffer-fallback-action` is empty).

Note also, that the combined action alist may contain duplicate entries and entries for the same key with different values. As a rule, action functions always use the first association of a key they find. Hence, the association an action function uses is not necessarily the association provided by the display action that specified that action function,

The argument *action* can also have a non-`nil`, non-list value. This has the special meaning that the buffer should be displayed in a window other than the selected one, even if the selected window is already displaying it. If called interactively with a prefix argument, *action* is `t`. Lisp programs should always supply a list value.

The optional argument *frame*, if non-`nil`, specifies which frames to check when deciding whether the buffer is already displayed. It is equivalent to adding an element `(reusable-frames . frame)` to the action alist of *action* (see Section 29.13.3 [Buffer Display Action Alists], page 733). The *frame* argument is provided for compatibility reasons, Lisp programs should not use it.

`display-buffer-overriding-action` [Variable]
The value of this variable should be a display action, which is treated with the highest priority by `display-buffer`. The default value is an empty display action, i.e., `(nil . nil)`.

`display-buffer-alist` [User Option]
The value of this option is an alist mapping conditions to display actions. Each condition is passed to `buffer-match-p` (see Section 28.8 [Buffer List], page 683), along with the buffer name and the *action* argument passed to `display-buffer`. If it returns a non-`nil` value, then `display-buffer` uses the corresponding display action to display the buffer. Caveat: if you use `derived-mode` or `major-mode` as condition, `buffer-match-p` could fail to report a match if `display-buffer` is called before the major mode of the buffer is set.

`display-buffer-base-action` [User Option]
The value of this option should be a display action. This option can be used to define a standard display action for calls to `display-buffer`.

`display-buffer-fallback-action` [Constant]
This display action specifies the fallback behavior for `display-buffer` if no other display actions are given.

29.13.2 Action Functions for Buffer Display

An *action function* is a function `display-buffer` calls for choosing a window to display a buffer. Action functions take two arguments: *buffer*, the buffer to display, and *alist*, an action alist (see Section 29.13.3 [Buffer Display Action Alists], page 733). They are supposed to return a window displaying *buffer* if they succeed and `nil` if they fail.

The following basic action functions are defined in Emacs.

`display-buffer-same-window` *buffer alist* [Function]
This function tries to display *buffer* in the selected window. It fails if the selected window is a minibuffer window or is dedicated to another buffer (see Section 29.15 [Dedicated Windows], page 751). It also fails if *alist* has a non-`nil` `inhibit-same-window` entry.

`display-buffer-reuse-window` *buffer alist* [Function]
This function tries to display *buffer* by finding a window that is already displaying it. Windows on the selected frame are preferred to windows on other frames.

If *alist* has a non-`nil` `inhibit-same-window` entry, the selected window is not eligible for reuse. The set of frames to search for a window already displaying *buffer* can be specified with the help of the `reusable-frames` action alist entry. If *alist* contains no `reusable-frames` entry, this function searches just the selected frame.

If this function chooses a window on another frame, it makes that frame visible and, unless *alist* contains an `inhibit-switch-frame` entry, raises that frame if necessary.

`display-buffer-reuse-mode-window` *buffer alist* [Function]
This function tries to display *buffer* by finding a window that is displaying a buffer in a given mode.

If *alist* contains a `mode` entry, its value specifies a major mode (a symbol) or a list of major modes. If *alist* contains no `mode` entry, the current major mode of *buffer* is used instead. A window is a candidate if it displays a buffer whose mode derives from one of the modes specified thusly.

The behavior is also controlled by *alist* entries for `inhibit-same-window`, `reusable-frames` and `inhibit-switch-frame`, like `display-buffer-reuse-window` does.

`display-buffer-pop-up-window` *buffer alist* [Function]
This function tries to display *buffer* by splitting the largest or least recently-used window (usually located on the selected frame). It actually performs the split by calling the function specified by `split-window-preferred-function` (see Section 29.13.4 [Choosing Window Options], page 738).

The size of the new window can be adjusted by supplying `window-height` and `window-width` entries in *alist*. If *alist* contains a `preserve-size` entry, Emacs will also try to preserve the size of the new window during future resize operations (see Section 29.6 [Preserving Window Sizes], page 709).

This function fails if no window can be split. More often than not, this happens because no window is large enough to allow splitting. Setting `split-height-threshold` or `split-width-threshold` to lower values may help in this regard. Splitting also fails when the selected frame has an `unsplittable` frame parameter; see Section 30.4.3.5 [Buffer Parameters], page 814.

`display-buffer-in-previous-window` *buffer alist* [Function]
This function tries to display *buffer* in a window where it was displayed previously.

If *alist* contains a non-`nil` `inhibit-same-window` entry, the selected window is not eligible for use. A dedicated window is usable only if it already shows *buffer*. If *alist* contains a `previous-window` entry, the window specified by that entry is usable even if it never showed *buffer* before.

If *alist* contains a `reusable-frames` entry (see Section 29.13.3 [Buffer Display Action Alists], page 733), its value determines which frames to search for a suitable window. If *alist* contains no `reusable-frames` entry, this function searches just the selected frame if `display-buffer-reuse-frames` and `pop-up-frames` are both `nil`; it searches all frames on the current terminal if either of those variables is non-`nil`.

If more than one window qualifies as usable according to these rules, this function makes a choice in the following order of preference:

- The window specified by any `previous-window` *alist* entry, provided it is not the selected window.
- A window that showed *buffer* before, provided it is not the selected window.

- The selected window if it is either specified by a `previous-window` *alist* entry or showed *buffer* before.

`display-buffer-use-some-window` *buffer alist* [Function]
This function tries to display *buffer* by choosing an existing window and displaying the buffer in that window. It first tries to find a window that has not been used recently (see Section 29.10 [Cyclic Window Ordering], page 719) on any frame specified by a `lru-frames` *alist* entry, falling back to the selected frame if no such entry exists. It also prefers windows that satisfy the constraints specified by `window-min-width` and `window-min-height` *alist* entries; preferring full-width windows if no `window-min-width` entry is found. Finally, it will not return a window whose use time is higher than that specified by any `lru-time` entry provided by *alist*.

If no less recently used window is found, this function will try to use some other window, preferably a large window on some visible frame. It can fail if all windows are dedicated to other buffers (see Section 29.15 [Dedicated Windows], page 751).

`display-buffer-use-least-recent-window` *buffer alist* [Function]
This function is similar to `display-buffer-use-some-window`, but will try harder to not use the a recently used window. In particular, it does not use the selected window. In addition, it will first try to reuse a window that shows *buffer* already, base the decision whether it should use a window showing another buffer on that window's use time alone and pop up a new window if no usable window is found.

Finally, this function will bump the use time (see Section 29.3 [Selecting Windows], page 698) of any window it returns in order to avoid that further invocations will use that window for showing another buffer. An application that wants to display several buffers in a row can help this function by providing a `lru-time` *alist* entry it has initially set to the value of the selected window's use time. Each invocation of this function will then bump the use time of the window returned to a value higher than that and a subsequent invocation will inhibit this function to use a window it returned earlier.

`display-buffer-in-direction` *buffer alist* [Function]
This function tries to display *buffer* at a location specified by *alist*. For this purpose, *alist* should contain a `direction` entry whose value is one of `left`, `above` (or `up`), `right` and `below` (or `down`). Other values are usually interpreted as `below`.

If *alist* also contains a `window` entry, its value specifies a reference window. That value can be a special symbol like `main` which stands for the selected frame's main window (see Section 29.17.2 [Side Window Options and Functions], page 755) or `root` standing for the selected frame's root window (see Section 29.2 [Windows and Frames], page 694). It can also specify an arbitrary valid window. Any other value (or omitting the `window` entry entirely) means to use the selected window as reference window.

This function first tries to reuse a window in the specified direction that already shows *buffer*. If no such window exists, it tries to split the reference window in order to produce a new window in the specified direction. If this fails as well, it will try to display *buffer* in an existing window in the specified direction. In either case,

the window chosen will appear on the side of the reference window specified by the `direction` entry, sharing at least one edge with the reference window.

If the reference window is live, the edge the chosen window will share with it is always the opposite of the one specified by the `direction` entry. For example, if the value of the `direction` entry is `left`, the chosen window's right edge coordinate (see Section 29.24 [Coordinates and Windows], page 772) will equal the reference window's left edge coordinate.

If the reference window is internal, a reused window must share with it the edge specified by the `direction` entry. Hence if, for example, the reference window is the frame's root window and the value of the `direction` entry is `left`, a reused window must be on the left of the frame. This means that the left edge coordinate of the chosen window and that of the reference window are the same.

A new window, however, will be created by splitting the reference window such that the chosen window will share the opposite edge with the reference window. In our example, a new root window would be created with a new live window and the reference window as its children. The chosen window's right edge coordinate would then equal the left edge coordinate of the reference window. Its left edge coordinate would equal the left edge coordinate of the frame's new root window.

Four special values for `direction` entries allow to implicitly specify the selected frame's main window as the reference window: `leftmost`, `top`, `rightmost` and `bottom`. This means that instead of, for example, `(direction . left) (window . main)` one can just specify `(direction . leftmost)`. An existing `window` *alist* entry is ignored in such cases.

`display-buffer-below-selected` *buffer alist* [Function]

This function tries to display *buffer* in a window below the selected window. If there is a window below the selected one and that window already displays *buffer*, it reuses that window.

If there is no such window, this function tries to create a new window by splitting the selected one, and displays *buffer* there. It will also try to adjust that window's size provided *alist* contains a suitable `window-height` or `window-width` entry, see above.

If splitting the selected window fails and there is a non-dedicated window below the selected one showing some other buffer, this function tries to use that window for showing *buffer*.

If *alist* contains a `window-min-height` entry, this function ensures that the window used is or can become at least as high as specified by that entry's value. Note that this is only a guarantee. In order to actually resize the window used, *alist* must also provide an appropriate `window-height` entry.

`display-buffer-at-bottom` *buffer alist* [Function]

This function tries to display *buffer* in a window at the bottom of the selected frame.

This either tries to split the window at the bottom of the frame or the frame's root window, or to reuse an existing window at the bottom of the selected frame.

`display-buffer-pop-up-frame` *buffer alist* [Function]
This function creates a new frame, and displays the buffer in that frame's window. It actually performs the frame creation by calling the function specified in `pop-up-frame-function` (see Section 29.13.4 [Choosing Window Options], page 738). If *alist* contains a `pop-up-frame-parameters` entry, the associated value is added to the newly created frame's parameters.

`display-buffer-full-frame` *buffer alist* [Function]
This function displays the buffer on the current frame, deleting all other windows so that it takes up the full frame.

`display-buffer-in-child-frame` *buffer alist* [Function]
This function tries to display *buffer* in a child frame (see Section 30.14 [Child Frames], page 833) of the selected frame, either reusing an existing child frame or by making a new one. If *alist* has a non-`nil` `child-frame-parameters` entry, the corresponding value is an alist of frame parameters to give the new frame. A `parent-frame` parameter specifying the selected frame is provided by default. If the child frame should become the child of another frame, a corresponding entry must be added to *alist*.

The appearance of child frames is largely dependent on the parameters provided via *alist*. It is advisable to use at least ratios to specify the size (see Section 30.4.3.3 [Size Parameters], page 810) and the position (see Section 30.4.3.2 [Position Parameters], page 807) of the child frame, and to add a `keep-ratio` parameter (see Section 30.4.3.6 [Frame Interaction Parameters], page 814), in order to make sure that the child frame remains visible. For other parameters that should be considered see Section 30.14 [Child Frames], page 833.

`display-buffer-use-some-frame` *buffer alist* [Function]
This function tries to display *buffer* by finding a frame that meets a predicate (by default any frame other than the selected frame).

If this function chooses a window on another frame, it makes that frame visible and, unless *alist* contains an `inhibit-switch-frame` entry, raises that frame if necessary.

If *alist* has a non-`nil` `frame-predicate` entry, its value is a function taking one argument (a frame), returning non-`nil` if the frame is a candidate; this function replaces the default predicate.

If *alist* has a non-`nil` `inhibit-same-window` entry, the selected window is not used; thus if the selected frame has a single window, it is not used.

`display-buffer-no-window` *buffer alist* [Function]
If *alist* has a non-`nil` `allow-no-window` entry, then this function does not display *buffer* and returns the symbol `fail`. This constitutes the only exception to the convention that an action function returns either `nil` or a window showing *buffer*. If *alist* has no such `allow-no-window` entry, this function returns `nil`.

If this function returns `fail`, `display-buffer` will skip the execution of any further display actions and return `nil` immediately. If this function returns `nil`, `display-buffer` will continue with the next display action, if any.

It is assumed that when a caller of `display-buffer` specifies a non-`nil` `allow-no-window` entry, it is also able to handle a `nil` return value.

Two other action functions are described in their proper sections—`display-buffer-in-side-window` (see Section 29.17.1 [Displaying Buffers in Side Windows], page 754) and `display-buffer-in-atom-window` (see Section 29.18 [Atomic Windows], page 759).

29.13.3 Action Alists for Buffer Display

An *action alist* is an association list mapping predefined symbols recognized by action functions to values these functions are supposed to interpret accordingly. In each call, `display-buffer` constructs a new, possibly empty action alist and passes that entire list on to any action function it calls.

By design, action functions are free in their interpretation of action alist entries. In fact, some entries like `allow-no-window` or `previous-window` have a meaning only for one or a few action functions, and are ignored by the rest. Other entries, like `inhibit-same-window` or `window-parameters`, are supposed to be respected by most action functions, including those provided by application programs and external packages.

In the previous subsection we have described in detail how individual action functions interpret the action alist entries they care about. Here we give a reference list of all known action alist entries according to their symbols, together with their values and action functions (see Section 29.13.2 [Buffer Display Action Functions], page 728) that recognize them. Throughout this list, the terms "buffer" will refer to the buffer `display-buffer` is supposed to display, and "value" refers to the entry's value.

`inhibit-same-window`
: If the value is non-`nil`, this signals that the selected window must not be used for displaying the buffer. All action functions that (re-)use an existing window should respect this entry.

`previous-window`
: The value must specify a window that may have displayed the buffer previously. `display-buffer-in-previous-window` will give preference to such a window provided it is still live and not dedicated to another buffer.

`mode`
: The value is either a major mode or a list of major modes. `display-buffer-reuse-mode-window` may reuse a window whenever the value specified by this entry matches the major mode of that window's buffer. Other action functions ignore such entries.

`frame-predicate`
: The value must be a function taking one argument (a frame), supposed to return non-`nil` if that frame is a candidate for displaying the buffer. This entry is used by `display-buffer-use-some-frame`.

`reusable-frames`
: The value specifies the set of frames to search for a window that can be reused because it already displays the buffer. It can be set as follows:

  - `nil` means consider only windows on the selected frame. (Actually, the last frame used that is not a minibuffer-only frame.)
  - `visible` means consider windows on all visible frames.
  - 0 means consider windows on all visible or iconified frames.

- A frame means consider windows on that frame only.
- `t` means consider windows on all frames. (Note that this value is rarely the right thing to use—it might also return a tooltip frame.)

Note that the meaning of `nil` differs slightly from that of the *all-frames* argument to `next-window` (see Section 29.10 [Cyclic Window Ordering], page 719).

A major client of this is `display-buffer-reuse-window`, but all other action functions that try to reuse a window are affected as well. `display-buffer-in-previous-window` consults it when searching for a window that previously displayed the buffer on another frame.

`inhibit-switch-frame`
: A non-`nil` value prevents another frame from being raised or selected, if the window chosen by `display-buffer` is displayed there. Primarily affected by this are `display-buffer-use-some-frame` and `display-buffer-reuse-window`. Ideally, `display-buffer-pop-up-frame` should be affected as well, but there is no guarantee that the window manager will comply.

`window-parameters`
: The value specifies an alist of window parameters to give the chosen window. All action functions that choose a window should process this entry.

`window-min-width`
: The value specifies a minimum width of the window used, in canonical frame columns. The special value `full-width` means the chosen window should be one that has no other windows on the left or right of it in its frame.

  This entry is currently honored by `display-buffer-use-some-window` and `display-buffer-use-least-recent-window`, which try hard to avoid returning a less recently used window that does not satisfy the entry.

  Note that providing such an entry alone does not necessarily make the window as wide as specified by its value. To actually resize an existing window or make a new window as wide as specified by this entry's value, a `window-width` entry specifying that value should be provided as well. Such a `window-width` entry can, however, specify a completely different value, or ask the window width to fit that of its buffer, in which case the `window-min-width` entry provides the guaranteed minimum width of the window.

`window-min-height`
: The value specifies a minimum height of the window used, in canonical frame lines. The special value `full-height` means the chosen window should be a full-height window, one that has no other windows above or below it in its frame.

  This entry is currently honored by `display-buffer-below-selected` which does not use a window that is not as high as specified by this entry. It's also honored by `display-buffer-use-some-window` and `display-buffer-use-least-recent-window` which try hard to avoid returning a less recently used window if it does not satisfy this constraint.

  Note that providing such an entry alone does not necessarily make the window as tall as specified by its value. To actually resize an existing window or make

a new window as tall as specified by that value, a `window-height` entry specifying that value should be provided as well. Such a `window-height` entry can, however, specify a completely different value or ask the window height to be fit to that of its buffer in which case the `window-min-height` entry provides the guaranteed minimum height of the window used.

`window-height`
: The value specifies whether and how to adjust the height of the chosen window and can be one of the following:

  - `nil` means to leave the height of the chosen window alone.
  - An integer number specifies the desired total height of the chosen window in lines.
  - A floating-point number specifies the fraction of the chosen window's desired total height with respect to the total height of its frame's root window.
  - A cons cell whose CAR is `body-lines` and whose CDR is an integer that specifies the height of the chosen window's body in frame lines.
  - If the value specifies a function, that function is called with one argument—the chosen window. The function is supposed to adjust the height of the window; its return value is ignored. Suitable functions are `fit-window-to-buffer` and `shrink-window-if-larger-than-buffer`, see Section 29.5 [Resizing Windows], page 705.

  By convention, the height of the chosen window is adjusted only if the window is part of a vertical combination (see Section 29.2 [Windows and Frames], page 694) to avoid changing the height of other, unrelated windows. Also, this entry should be processed only under certain conditions which are specified right below this list.

`window-width`
: This entry is similar to the `window-height` entry described before, but used to adjust the chosen window's width instead. The value can be one of the following:

  - `nil` means to leave the width of the chosen window alone.
  - An integer specifies the desired total width of the chosen window in columns.
  - A floating-point number specifies the fraction of the chosen window's desired total width with respect to the total width of the frame's root window.
  - A cons cell whose CAR is `body-columns` and whose CDR is an integer that specifies the width of the chosen window's body in frame columns.
  - If the value specifies a function, that function is called with one argument—the chosen window. The function is supposed to adjust the width of the window; its return value is ignored.

`window-size`
: This entry is a combination of the two preceding ones and can be used to adjust the chosen window's height *and* width. Since windows can be resized in one direction only without affecting other windows, `window-size` is effective only

to set up the size of a window appearing alone on a frame. The value can be one of the following:

- `nil` means to leave the size of the chosen window alone.
- A cons cell of two integers specifies the desired total width and height of the chosen window in lines and columns. It's effect is to adjust the size of the frame accordingly.
- A cons cell whose CAR equals `body-chars` and whose CDR is a cons cell of two integers—the desired body width and height of the chosen window in frame columns and lines. It's effect is to adjust the size of the frame accordingly.
- If the value specifies a function, that function is called with one argument—the chosen window. The function is supposed to adjust the size of the window's frame; its return value is ignored.

This entry should be processed under only certain conditions which are specified right below this list.

`dedicated`
: If non-`nil`, such an entry tells `display-buffer` to mark any window it creates as dedicated to its buffer (see Section 29.15 [Dedicated Windows], page 751). It does that by calling `set-window-dedicated-p` with the chosen window as first argument and the entry's value as second. Side windows are by default dedicated with the value `side` (see Section 29.17.2 [Side Window Options and Functions], page 755).

`preserve-size`
: If non-`nil` such an entry tells Emacs to preserve the size of the window chosen (see Section 29.6 [Preserving Window Sizes], page 709). The value should be either `(t . nil)` to preserve the width of the window, `(nil . t)` to preserve its height or `(t . t)` to preserve both, its width and its height. This entry should be processed only under certain conditions which are specified right after this list.

`lru-frames`
: The value specifies the set of frames to search for a window that can be used to display the buffer. It is honored by `display-buffer-use-some-window` and `display-buffer-use-least-recent-window` when trying to find a less recently used window showing some other buffer. Its values are the same as for the `reusable-frames` entry described above.

`lru-time` The value is supposed to specify a use time (see Section 29.3 [Selecting Windows], page 698). This entry is honored by `display-buffer-use-some-window` and `display-buffer-use-least-recent-window` when trying to find a less recently used window showing some other buffer. If a window's use time is higher than the value specified by this option, these action functions will not consider such a window for displaying the buffer.

`bump-use-time`
: If non-`nil`, such an entry will cause `display-buffer` to bump the use time (see Section 29.3 [Selecting Windows], page 698) of the window it uses. This

should avoid later use of this window by action functions like `display-buffer-use-some-window` and `display-buffer-use-least-recent-window` for showing another buffer.

There is a fine difference between using this entry and using the action function `display-buffer-use-least-recent-window`. Calling the latter means to only bump the use times of windows that function uses for displaying the buffer. The entry described here will cause `display-buffer` to bump the use time of *any* window used for displaying a buffer.

`pop-up-frame-parameters`
: The value specifies an alist of frame parameters to give a new frame, if one is created. `display-buffer-pop-up-frame` is its one and only addressee.

`parent-frame`
: The value specifies the parent frame to be used when the buffer is displayed on a child frame. This entry is used only by `display-buffer-in-child-frame`.

`child-frame-parameters`
: The value specifies an alist of frame parameters to use when the buffer is displayed on a child frame. This entry is used only by `display-buffer-in-child-frame`.

`side`
: The value denotes the side of the frame or window where a new window displaying the buffer shall be created. This entry is used by `display-buffer-in-side-window` to indicate the side of the frame where a new side window shall be placed (see Section 29.17.1 [Displaying Buffers in Side Windows], page 754). It is also used by `display-buffer-in-atom-window` to indicate the side of an existing window where the new window shall be located (see Section 29.18 [Atomic Windows], page 759).

`slot`
: If non-`nil`, the value specifies the slot of the side window supposed to display the buffer. This entry is used only by `display-buffer-in-side-window`.

`direction`
: The value specifies a direction which, together with a `window` entry, allows `display-buffer-in-direction` to determine the location of the window to display the buffer.

`window`
: The value specifies a window that is in some way related to the window chosen by `display-buffer`. This entry is currently used by `display-buffer-in-atom-window` to indicate the window on whose side the new window shall be created. It is also used by `display-buffer-in-direction` to specify the reference window on whose side the resulting window shall appear.

`allow-no-window`
: If the value is non-`nil`, `display-buffer` does not necessarily have to display the buffer and the caller is prepared to accept that. This entry is not intended for user customizations, since there is no guarantee that an arbitrary caller of `display-buffer` will be able to handle the case that no window will display the buffer. `display-buffer-no-window` is the only action function that cares about this entry.

`body-function`
: The value must be a function taking one argument (a displayed window). This function can be used to fill the displayed window's body with some contents that might depend on dimensions of the displayed window. It is called *after* the buffer is displayed, and *before* the entries `window-height`, `window-width` and `preserve-size` are applied that could resize the window to fit it to the inserted contents.

By convention, the entries `window-height`, `window-width` and `preserve-size` are applied after the chosen window's buffer has been set up and if and only if that window never showed another buffer before. More precisely, the latter means that the window must have been either created by the current `display-buffer` call or the window was created earlier by `display-buffer` to show the buffer and never was used to show another buffer until it was reused by the current invocation of `display-buffer`.

If no `window-height`, `window-width` or `window-size` entry was specified, the window may still be resized automatically when the buffer is temporary and `temp-buffer-resize-mode` has been enabled, Section 41.8 [Temporary Displays], page 1150. In that case, the CDR of a `window-height`, `window-width` or `window-size` entry can be used to inhibit or override the default behavior of `temp-buffer-resize-mode` for specific buffers or invocations of `display-buffer`.

29.13.4 Additional Options for Displaying Buffers

The behavior of buffer display actions (see Section 29.13.1 [Choosing Window], page 726) can be further modified by the following user options.

`pop-up-windows` [User Option]
: If the value of this variable is non-`nil`, `display-buffer` is allowed to split an existing window to make a new window for displaying in. This is the default.

  This variable is provided for backward compatibility only. It is obeyed by `display-buffer` via a special mechanism in `display-buffer-fallback-action`, which calls the action function `display-buffer-pop-up-window` (see Section 29.13.2 [Buffer Display Action Functions], page 728) when the value of this option is non-`nil`. It is not consulted by `display-buffer-pop-up-window` itself, which the user may specify directly in `display-buffer-alist` etc.

`split-window-preferred-function` [User Option]
: This variable specifies a function for splitting a window, in order to make a new window for displaying a buffer. It is used by the `display-buffer-pop-up-window` action function to actually split the window.

  The value must be a function that takes one argument, a window, and returns either a new window (which will be used to display the desired buffer) or `nil` (which means the splitting failed). The default value is `split-window-sensibly`, which is documented next.

`split-window-sensibly` **&optional** *window* [Function]
: This function tries to split *window* and return the newly created window. If *window* cannot be split, it returns `nil`. If *window* is omitted or `nil`, it defaults to the selected window.

This function obeys the usual rules that determine when a window may be split (see Section 29.7 [Splitting Windows], page 710). It first tries to split by placing the new window below, subject to the restriction imposed by `split-height-threshold` (see below), in addition to any other restrictions. If that fails, it tries to split by placing the new window to the right, subject to `split-width-threshold` (see below). If that also fails, and the window is the only window on its frame, this function again tries to split and place the new window below, disregarding `split-height-threshold`. If this fails as well, this function gives up and returns `nil`.

`split-height-threshold` [User Option]
This variable specifies whether `split-window-sensibly` is allowed to split the window placing the new window below. If it is an integer, that means to split only if the original window has at least that many lines. If it is `nil`, that means not to split this way.

`split-width-threshold` [User Option]
This variable specifies whether `split-window-sensibly` is allowed to split the window placing the new window to the right. If the value is an integer, that means to split only if the original window has at least that many columns. If the value is `nil`, that means not to split this way.

`even-window-sizes` [User Option]
This variable, if non-`nil`, causes `display-buffer` to even window sizes whenever it reuses an existing window, and that window is adjacent to the selected one.

If its value is `width-only`, sizes are evened only if the reused window is on the left or right of the selected one and the selected window is wider than the reused one. If its value is `height-only` sizes are evened only if the reused window is above or beneath the selected window and the selected window is higher than the reused one. Any other non-`nil` value means to even sizes in any of these cases provided the selected window is larger than the reused one in the sense of their combination.

`pop-up-frames` [User Option]
If the value of this variable is non-`nil`, that means `display-buffer` may display buffers by making new frames. The default is `nil`.

A non-`nil` value also means that when `display-buffer` is looking for a window already displaying *buffer-or-name*, it can search any visible or iconified frame, not just the selected frame.

This variable is provided mainly for backward compatibility. It is obeyed by `display-buffer` via a special mechanism in `display-buffer-fallback-action`, which calls the action function `display-buffer-pop-up-frame` (see Section 29.13.2 [Buffer Display Action Functions], page 728) if the value is non-`nil`. (This is done before attempting to split a window.) This variable is not consulted by `display-buffer-pop-up-frame` itself, which the user may specify directly in `display-buffer-alist` etc.

`pop-up-frame-function` [User Option]
This variable specifies a function for creating a new frame, in order to make a new window for displaying a buffer. It is used by the `display-buffer-pop-up-frame` action function.

The value should be a function that takes no arguments and returns a frame, or `nil` if no frame could be created. The default value is a function that creates a frame using the parameters specified by `pop-up-frame-alist` (see below).

`pop-up-frame-alist` [User Option]

This variable holds an alist of frame parameters (see Section 30.4 [Frame Parameters], page 804), which is used by the function specified by `pop-up-frame-function` to make a new frame. The default is `nil`.

This option is provided for backward compatibility only. Note, that when `display-buffer-pop-up-frame` calls the function specified by `pop-up-frame-function`, it prepends the value of all `pop-up-frame-parameters` action alist entries to `pop-up-frame-alist` so that the values specified by the action alist entry effectively override any corresponding values of `pop-up-frame-alist`.

Hence, users should set up a `pop-up-frame-parameters` action alist entry in `display-buffer-alist` instead of customizing `pop-up-frame-alist`. Only this will guarantee that the value of a parameter specified by the user overrides the value of that parameter specified by the caller of `display-buffer`.

Many efforts in the design of `display-buffer` have been given to maintain compatibility with code that uses older options like `pop-up-windows`, `pop-up-frames`, `pop-up-frame-alist`, `same-window-buffer-names` and `same-window-regexps`. Lisp Programs and users should refrain from using these options. Above we already warned against customizing `pop-up-frame-alist`. Here we describe how to convert the remaining options to use display actions instead.

`pop-up-windows`

This variable is `t` by default. Instead of customizing it to `nil` and thus telling `display-buffer` what not to do, it's much better to list in `display-buffer-base-action` the action functions it should try instead as, for example:

```
(setopt
 display-buffer-base-action
 '((display-buffer-reuse-window display-buffer-same-window
    display-buffer-in-previous-window
    display-buffer-use-some-window)))
```

`pop-up-frames`

Instead of customizing this variable to `t`, customize `display-buffer-base-action`, for example, as follows:

```
(setopt
 display-buffer-base-action
 '((display-buffer-reuse-window display-buffer-pop-up-frame)
   (reusable-frames . 0)))
```

`same-window-buffer-names`
`same-window-regexps`

Instead of adding a buffer name or a regular expression to one of these options use a `display-buffer-alist` entry for that buffer specifying the action function `display-buffer-same-window`.

```
(setopt
 display-buffer-alist
 (cons '("\\*foo\\*" (display-buffer-same-window))
       display-buffer-alist))
```

29.13.5 Precedence of Action Functions

From the past subsections we already know that `display-buffer` must be supplied with a number of display actions (see Section 29.13.1 [Choosing Window], page 726) in order to display a buffer. In a completely uncustomized Emacs, these actions are specified by `display-buffer-fallback-action` in the following order of precedence: Reuse a window, pop up a new window on the same frame, use a window previously showing the buffer, use some window and pop up a new frame. (Note that the remaining actions named by `display-buffer-fallback-action` are void in an uncustomized Emacs).

Consider the following form:

```
(display-buffer (get-buffer-create "*foo*"))
```

Evaluating this form in the buffer `*scratch*` of an uncustomized Emacs session will usually fail to reuse a window that shows `*foo*` already, but succeed in popping up a new window. Evaluating the same form again will now not cause any visible changes—`display-buffer` reused the window already showing `*foo*` because that action was applicable and had the highest precedence among all applicable actions.

Popping up a new window will fail if there is not enough space on the selected frame. In an uncustomized Emacs it typically fails when there are already two windows on a frame. For example, if you now type *C-x 1* followed by *C-x 2* and evaluate the form once more, `*foo*` should show up in the lower window—`display-buffer` just used "some" window. If, before typing *C-x 2* you had typed *C-x o*, `*foo*` would have been shown in the upper window because "some" window stands for the "least recently used" window and the selected window has been least recently used if and only if it is alone on its frame.

Let's assume you did not type *C-x o* and `*foo*` is shown in the lower window. Type *C-x o* to get there followed by *C-x left* and evaluate the form again. This should display `*foo*` in the same, lower window because that window had already shown `*foo*` previously and was therefore chosen instead of some other window.

So far we have only observed the default behavior in an uncustomized Emacs session. To see how this behavior can be customized, let's consider the option `display-buffer-base-action`. It provides a very coarse customization which conceptually affects the display of *any* buffer. It can be used to supplement the actions supplied by `display-buffer-fallback-action` by reordering them or by adding actions that are not present there but fit more closely the user's editing practice. However, it can also be used to change the default behavior in a more profound way.

Let's consider a user who, as a rule, prefers to display buffers on another frame. Such a user might provide the following customization:

```
(setopt
 display-buffer-base-action
 '((display-buffer-reuse-window display-buffer-pop-up-frame)
   (reusable-frames . 0)))
```

This setting will cause `display-buffer` to first try to find a window showing the buffer on a visible or iconified frame and, if no such frame exists, pop up a new frame. You can observe this behavior on a graphical system by typing `C-x 1` in the window showing `*scratch*` and evaluating our canonical `display-buffer` form. This will usually create (and give focus to) a new frame whose root window shows `*foo*`. Iconify that frame and evaluate the canonical form again: `display-buffer` will reuse the window on the new frame (usually raising the frame and giving it focus too).

Only if creating a new frame fails, `display-buffer` will apply the actions supplied by `display-buffer-fallback-action` which means to again try reusing a window, popping up a new window and so on. A trivial way to make frame creation fail is supplied by the following form:

```
(let ((pop-up-frame-function 'ignore))
  (display-buffer (get-buffer-create "*foo*")))
```

We will forget about that form immediately after observing that it fails to create a new frame and uses a fallback action instead.

Note that `display-buffer-reuse-window` appears redundant in the customization of `display-buffer-base-action` because it is already part of `display-buffer-fallback-action` and should be tried there anyway. However, that would fail because due to the precedence of `display-buffer-base-action` over `display-buffer-fallback-action`, at that time `display-buffer-pop-up-frame` would have already won the race. In fact, this:

```
(setopt
 display-buffer-base-action
 '(display-buffer-pop-up-frame (reusable-frames . 0)))
```

would cause `display-buffer` to *always* pop up a new frame which is probably not what our user wants.

So far, we have only shown how *users* can customize the default behavior of `display-buffer`. Let us now see how *applications* can change the course of `display-buffer`. The canonical way to do that is to use the *action* argument of `display-buffer` or a function that calls it, like, for example, `pop-to-buffer` (see Section 29.12 [Switching Buffers], page 723).

Suppose an application wants to display `*foo*` preferably below the selected window (to immediately attract the attention of the user to the new window) or, if that fails, in a window at the bottom of the frame. It could do that with a call like this:

```
(display-buffer
 (get-buffer-create "*foo*")
 '((display-buffer-below-selected display-buffer-at-bottom)))
```

In order to see how this new, modified form works, delete any frame showing `*foo*`, type `C-x 1` followed by `C-x 2` in the window showing `*scratch*`, and subsequently evaluate that form. `display-buffer` should split the upper window, and show `*foo*` in the new window. Alternatively, if after `C-x 2` you had typed `C-x o`, `display-buffer` would have split the window at the bottom instead.

Suppose now that, before evaluating the new form, you have made the selected window as small as possible, for example, by evaluating the form `(fit-window-to-buffer)` in that window. In that case, `display-buffer` would have failed to split the selected window and

would have split the frame's root window instead, effectively displaying `*foo*` at the bottom of the frame.

In either case, evaluating the new form a second time should reuse the window already showing `*foo*` since both functions supplied by the *action* argument try to reuse such a window first.

By setting the *action* argument, an application effectively overrules any customization of `display-buffer-base-action`. Our user can now either accept the choice of the application, or redouble by customizing the option `display-buffer-alist` as follows:

```
(setopt
 display-buffer-alist
 '(("\\*foo\\*"
    (display-buffer-reuse-window display-buffer-pop-up-frame))))
```

Trying this with the new, modified form above in a configuration that does not show `*foo*` anywhere, will display `*foo*` on a separate frame, completely ignoring the *action* argument of `display-buffer`.

Note that we didn't care to specify a `reusable-frames` action alist entry in our specification of `display-buffer-alist`. `display-buffer` always takes the first one it finds—in our case the one specified by `display-buffer-base-action`. If we wanted to use a different specification, for example, to exclude iconified frames showing `*foo*` from the list of reusable ones, we would have to specify that separately, however:

```
(setopt
 display-buffer-alist
 '(("\\*foo\\*"
    (display-buffer-reuse-window display-buffer-pop-up-frame)
    (reusable-frames . visible))))
```

If you try this, you will notice that repeated attempts to display `*foo*` will succeed to reuse a frame only if that frame is visible.

The above example would allow the conclusion that users customize `display-buffer-alist` for the sole purpose to overrule the *action* argument chosen by applications. Such a conclusion would be incorrect. `display-buffer-alist` is the standard option for users to direct the course of display of specific buffers in a preferred way regardless of whether the display is also guided by an *action* argument.

We can, however, reasonably conclude that customizing `display-buffer-alist` differs from customizing `display-buffer-base-action` in two major aspects: it is stronger because it overrides the *action* argument of `display-buffer`, and it allows to explicitly specify the affected buffers. In fact, displaying other buffers is not affected in any way by a customization for `*foo*`. For example,

```
(display-buffer (get-buffer-create "*bar*"))
```

continues being governed by the settings of `display-buffer-base-action` and `display-buffer-fallback-action` only.

We could stop with our examples here but Lisp programs still have an ace up their sleeves which they can use to overrule any customization of `display-buffer-alist`. It's the variable `display-buffer-overriding-action` which they can bind around `display-buffer` calls as follows:

```
(let ((display-buffer-overriding-action
       '((display-buffer-same-window))))
  (display-buffer
   (get-buffer-create "*foo*")
   '((display-buffer-below-selected display-buffer-at-bottom))))
```

Evaluating this form will usually display `*foo*` in the selected window regardless of the *action* argument and any user customizations. (Usually, an application will not bother to also provide an *action* argument. Here it just serves to illustrate the fact that it gets overridden.)

It might be illustrative to look at the list of action functions `display-buffer` would have tried to display `*foo*` with the customizations we provided here. The list (including comments explaining who added this and the subsequent elements) is:

```
(display-buffer-same-window  ;; `display-buffer-overriding-action'
 display-buffer-reuse-window ;; `display-buffer-alist'
 display-buffer-pop-up-frame
 display-buffer-below-selected ;; ACTION argument
 display-buffer-at-bottom
 display-buffer-reuse-window ;; `display-buffer-base-action'
 display-buffer-pop-up-frame
 display-buffer--maybe-same-window ;; `display-buffer-fallback-action'
 display-buffer-reuse-window
 display-buffer--maybe-pop-up-frame-or-window
 display-buffer-in-previous-window
 display-buffer-use-some-window
 display-buffer-pop-up-frame)
```

Note that among the internal functions listed here, `display-buffer--maybe-same-window` is effectively ignored while `display-buffer--maybe-pop-up-frame-or-window` actually runs `display-buffer-pop-up-window`.

The action alist passed in each function call is:

```
((reusable-frames . visible)
 (reusable-frames . 0))
```

which shows that we have used the second specification of `display-buffer-alist` above, overriding the specification supplied by `display-buffer-base-action`. Suppose our user had written that as

```
(setopt
 display-buffer-alist
 '(("\\*foo\\*"
    (display-buffer-reuse-window display-buffer-pop-up-frame)
    (inhibit-same-window . t)
    (reusable-frames . visible))))
```

In this case the `inhibit-same-window` alist entry will successfully invalidate the `display-buffer-same-window` specification from `display-buffer-overriding-action` and `display-buffer` will show `*foo*` on another frame. To make `display-buffer-overriding-action` more robust in this regard, the application would have to specify an appropriate `inhibit-same-window` entry too, for example, as follows:

```
(let ((display-buffer-overriding-action
       '(display-buffer-same-window (inhibit-same-window . nil))))
  (display-buffer (get-buffer-create "*foo*")))
```

This last example shows that while the precedence order of action functions is fixed, as described in Section 29.13.1 [Choosing Window], page 726, an action alist entry specified by a display action ranked lower in that order can affect the execution of a higher ranked display action.

29.13.6 The Zen of Buffer Display

In its most simplistic form, a frame accommodates always one single window that can be used for displaying a buffer. As a consequence, it is always the latest call of `display-buffer` that will have succeeded in placing its buffer there.

Since working with such a frame is not very practical, Emacs by default allows for more complex frame layouts controlled by the default values of the frame size and the `split-height-threshold` and `split-width-threshold` options. Displaying a buffer not yet shown on a frame then either splits the single window on that frame or (re-)uses one of its two windows.

The default behavior is abandoned as soon as the user customizes one of these thresholds or manually changes the frame's layout. The default behavior is also abandoned when calling `display-buffer` with a non-`nil` *action* argument or the user customizes one of the options mentioned in the previous subsections. Mastering `display-buffer` soon may become a frustrating experience due to the plethora of applicable display actions and the resulting frame layouts.

However, refraining from using buffer display functions and falling back on a split & delete windows metaphor is not a good idea either. Buffer display functions give Lisp programs and users a framework to reconcile their different needs; no comparable framework exists for splitting and deleting windows. Buffer display functions also allow to at least partially restore the layout of a frame when removing a buffer from it later (see Section 29.16 [Quitting Windows], page 752).

Below we will give a number of guidelines to redeem the frustration mentioned above and thus to avoid literally losing buffers in-between the windows of a frame.

Write display actions without stress

> Writing display actions can be a pain because one has to lump together action functions and action alists in one huge list. (Historical reasons prevented us from having `display-buffer` support separate arguments for these.) It might help to memorize some basic forms like the ones listed below:
>
> ```
> '(nil (inhibit-same-window . t))
> ```
>
> specifies an action alist entry only and no action function. Its sole purpose is to inhibit a `display-buffer-same-window` function specified elsewhere from showing the buffer in the same window, see also the last example of the preceding subsection.
>
> ```
> '(display-buffer-below-selected)
> ```
>
> on the other hand, specifies one action function and an empty action alist. To combine the effects of the above two specifications one would write the form
>
> ```
> '(display-buffer-below-selected (inhibit-same-window . t))
> ```

to add another action function one would write

```
'((display-buffer-below-selected display-buffer-at-bottom)
  (inhibit-same-window . t))
```

and to add another alist entry one would write

```
'((display-buffer-below-selected display-buffer-at-bottom)
  (inhibit-same-window . t)
  (window-height . fit-window-to-buffer))
```

That last form can be used as *action* argument of `display-buffer` in the following way:

```
(display-buffer
 (get-buffer-create "*foo*")
 '((display-buffer-below-selected display-buffer-at-bottom)
   (inhibit-same-window . t)
   (window-height . fit-window-to-buffer)))
```

In a customization of `display-buffer-alist` it would be used as follows:

```
(setopt
 display-buffer-alist
 '(("\\*foo\\*"
    (display-buffer-below-selected display-buffer-at-bottom)
    (inhibit-same-window . t)
    (window-height . fit-window-to-buffer))))
```

To add a customization for a second buffer one would then write:

```
(setopt
 display-buffer-alist
 '(("\\*foo\\*"
    (display-buffer-below-selected display-buffer-at-bottom)
    (inhibit-same-window . t)
    (window-height . fit-window-to-buffer))
   ("\\*bar\\*"
    (display-buffer-reuse-window display-buffer-pop-up-frame)
    (reusable-frames . visible))))
```

Treat each other with respect

`display-buffer-alist` and `display-buffer-base-action` are user options—Lisp programs must never set or rebind them. `display-buffer-overriding-action`, on the other hand, is reserved for applications—who seldom use that option and if they use it, then with utmost care.

Older implementations of `display-buffer` frequently caused users and applications to fight over the settings of user options like `pop-up-frames` and `pop-up-windows` (see Section 29.13.4 [Choosing Window Options], page 738). This was one major reason for redesigning `display-buffer`—to provide a clear framework specifying what users and applications should be allowed to do.

Lisp programs must be prepared that user customizations may cause buffers to get displayed in an unexpected way. They should never assume in their subsequent behavior, that the buffer has been shown precisely the way they asked for in the *action* argument of `display-buffer`.

Users should not pose too many and too severe restrictions on how arbitrary buffers get displayed. Otherwise, they will risk to lose the characteristics of showing a buffer for a certain purpose. Suppose a Lisp program has been written to compare different versions of a buffer in two windows side-by-side. If the customization of `display-buffer-alist` prescribes that any such buffer should be always shown in or below the selected window, the program will have a hard time to set up the desired window configuration via `display-buffer`.

To specify a preference for showing an arbitrary buffer, users should customize `display-buffer-base-action`. An example of how users who prefer working with multiple frames would do that was given in the previous subsection. `display-buffer-alist` should be reserved for displaying specific buffers in a specific way.

Consider reusing a window that already shows the buffer

Generally, it's always a good idea for users and Lisp programmers to be prepared for the case that a window already shows the buffer in question and to reuse that window. In the preceding subsection we have shown that failing to do so properly may cause `display-buffer` to continuously pop up a new frame although a frame showing that buffer existed already. In a few cases only, it might be undesirable to reuse a window, for example, when a different portion of the buffer should be shown in that window.

Hence, `display-buffer-reuse-window` is one action function that should be used as often as possible, both in *action* arguments and customizations. An `inhibit-same-window` entry in the *action* argument usually takes care of the most common case where reusing a window showing the buffer should be avoided—that where the window in question is the selected one.

Attract focus to the window chosen

This is a no-brainer for people working with multiple frames—the frame showing the buffer will automatically raise and get focus unless an `inhibit-switch-frame` entry forbids it. For single frame users this task can be considerably more difficult. In particular, `display-buffer-pop-up-window` and `display-buffer-use-some-window` can become obtrusive in this regard. They split or use a seemingly arbitrary (often the largest or least recently used) window, distracting the user's attention.

Some Lisp programs therefore try to choose a window at the bottom of the frame, for example, in order to display the buffer in vicinity of the minibuffer window where the user is expected to answer a question related to the new window. For non-input related actions `display-buffer-below-selected` might be preferable because the selected window usually already has the user's attention.

Take care which window is selected

Many applications call `display-buffer` from within window excursions produced by `with-selected-window` or `select-window` calls with a non-`nil` *norecord* argument. This is almost always a bad idea because the window selected within such an excursion is usually not the window selected in the configuration presented to the user.

If, for example, a user had added an `inhibit-same-window` alist entry, that entry would have avoided the window selected within the scope of the excursion and not the window selected in the resulting configuration. Even if no such entry has been added, the resulting behavior might be strange. While in a frame containing one live window, evaluating the following form

```
(progn
  (split-window)
  (display-buffer "*Messages*"))
```

will display a window showing the `*Messages*` buffer at the bottom and leave the other window selected. Evaluating the next form

```
(with-selected-window (split-window)
  (display-buffer "*Messages*"))
```

will display `*Messages*` in a window on the top and select it which is usually not what `display-buffer` is supposed to do.

On the other hand, while evaluating the following form

```
(progn
  (split-window)
  (pop-to-buffer "*Messages*"))
```

will correctly select the `*Messages*` buffer, the next form

```
(progn
  (split-window)
  (with-selected-window (selected-window)
    (pop-to-buffer "*Messages*")))
```

will not.

Also, invocations of action functions like `display-buffer-use-some-window` and `display-buffer-use-least-recent-window` that expect the selected window to have the highest use time among all windows, may fail to produce a window according to their specifications.

Hence, an application that relies on using a window excursion should try to postpone the `display-buffer` call until after the excursion has terminated.

29.14 Window History

Each window remembers in a list the buffers it has previously displayed, and the order in which these buffers were removed from it. This history is used, for example, by `replace-buffer-in-windows` (see Section 29.11 [Buffers and Windows], page 722), and when quitting windows (see Section 29.16 [Quitting Windows], page 752). The list is automatically maintained by Emacs, but you can use the following functions to explicitly inspect or alter it:

`window-prev-buffers` **&optional** *window* [Function]

This function returns a list specifying the previous contents of *window*. The optional argument *window* should be a live window and defaults to the selected one.

Each list element has the form `(`*`buffer window-start window-pos`*`)`, where *buffer* is a buffer previously shown in the window, *window-start* is the window start position

(see Section 29.20 [Window Start and End], page 762) when that buffer was last shown, and *window-pos* is the point position (see Section 29.19 [Window Point], page 761) when that buffer was last shown in *window*.

The list is ordered so that earlier elements correspond to more recently-shown buffers, and the first element usually corresponds to the buffer most recently removed from the window.

`set-window-prev-buffers` *window prev-buffers* [Function]

This function sets *window*'s previous buffers to the value of *prev-buffers*. The argument *window* must be a live window and defaults to the selected one. The argument *prev-buffers* should be a list of the same form as that returned by `window-prev-buffers`.

In addition, each window maintains a list of *next buffers*, which is a list of buffers re-shown by `switch-to-prev-buffer` (see below). This list is mainly used by `switch-to-prev-buffer` and `switch-to-next-buffer` for choosing buffers to switch to.

`window-next-buffers` **&optional** *window* [Function]

This function returns the list of buffers recently re-shown in *window* via `switch-to-prev-buffer`. The *window* argument must denote a live window or `nil` (meaning the selected window).

`set-window-next-buffers` *window next-buffers* [Function]

This function sets the next buffer list of *window* to *next-buffers*. The *window* argument should be a live window or `nil` (meaning the selected window). The argument *next-buffers* should be a list of buffers.

The following commands can be used to cycle through the global buffer list, much like `bury-buffer` and `unbury-buffer`. However, they cycle according to the specified window's history list, rather than the global buffer list. In addition, they restore window-specific window start and point positions, and may show a buffer even if it is already shown in another window. The `switch-to-prev-buffer` command, in particular, is used by `replace-buffer-in-windows`, `bury-buffer` and `quit-window` to find a replacement buffer for a window.

`switch-to-prev-buffer` **&optional** *window bury-or-kill* [Command]

This command displays the previous buffer in *window*. The argument *window* should be a live window or `nil` (meaning the selected window). If the optional argument *bury-or-kill* is non-`nil`, this means that the buffer currently shown in *window* is about to be buried or killed and consequently should not be switched to in future invocations of this command.

The previous buffer is usually the buffer shown before the buffer currently shown in *window*. However, a buffer that has been buried or killed, or has been already shown by a recent invocation of `switch-to-prev-buffer`, does not qualify as previous buffer.

If repeated invocations of this command have already shown all buffers previously shown in *window*, further invocations will show buffers from the buffer list of the frame *window* appears on (see Section 28.8 [Buffer List], page 683).

The option `switch-to-prev-buffer-skip` described below can be used to inhibit switching to certain buffers, for example, to those already shown in another window. Also, if *window*'s frame has a `buffer-predicate` parameter (see Section 30.4.3.5 [Buffer Parameters], page 814), that predicate may inhibit switching to certain buffers.

`switch-to-next-buffer` **&optional** *window* [Command]
This command switches to the next buffer in *window*, thus undoing the effect of the last `switch-to-prev-buffer` command in *window*. The argument *window* must be a live window and defaults to the selected one.

If there is no recent invocation of `switch-to-prev-buffer` that can be undone, this function tries to show a buffer from the buffer list of the frame *window* appears on (see Section 28.8 [Buffer List], page 683).

The option `switch-to-prev-buffer-skip` and the `buffer-predicate` (see Section 30.4.3.5 [Buffer Parameters], page 814) of *window*'s frame affect this command as they do for `switch-to-prev-buffer`.

By default `switch-to-prev-buffer` and `switch-to-next-buffer` can switch to a buffer that is already shown in another window. The following option can be used to override this behavior.

`switch-to-prev-buffer-skip` [User Option]
If this variable is `nil`, `switch-to-prev-buffer` may switch to any buffer, including those already shown in other windows.

If this variable is non-`nil`, `switch-to-prev-buffer` will refrain from switching to certain buffers. The following values can be used:

- `this` means do not switch to a buffer shown on the frame that hosts the window `switch-to-prev-buffer` is acting upon.
- `visible` means do not switch to a buffer shown on any visible frame.
- 0 (the number zero) means do not switch to a buffer shown on any visible or iconified frame.
- `t` means do not switch to a buffer shown on any live frame.
- A function that takes three arguments—the *window* argument of `switch-to-prev-buffer`, a buffer `switch-to-prev-buffer` intends to switch to and the *bury-or-kill* argument of `switch-to-prev-buffer`. If that function returns non-`nil`, `switch-to-prev-buffer` will refrain from switching to the buffer specified by the second argument.

The command `switch-to-next-buffer` obeys this option in a similar way. If this option specifies a function, `switch-to-next-buffer` will call that function with the third argument always `nil`.

Note that since `switch-to-prev-buffer` is called by `bury-buffer`, `replace-buffer-in-windows` and `quit-restore-window` as well, customizing this option may also affect the behavior of Emacs when a window is quit or a buffer gets buried or killed.

Note also that under certain circumstances `switch-to-prev-buffer` and `switch-to-next-buffer` may ignore this option, for example, when there is only one buffer left these functions can switch to.

`switch-to-prev-buffer-skip-regexp` [User Option]
: This user option should be either a regular expression or a list of regular expressions. Buffers whose names match one of those regular expressions will be ignored by `switch-to-prev-buffer` and `switch-to-next-buffer` (except when there's no other buffer to switch to).

29.15 Dedicated Windows

Functions for displaying a buffer can be told to not use specific windows by marking these windows as *dedicated* to their buffers. `display-buffer` (see Section 29.13.1 [Choosing Window], page 726) never uses a dedicated window for displaying another buffer in it. `get-lru-window` and `get-largest-window` (see Section 29.10 [Cyclic Window Ordering], page 719) do not consider dedicated windows as candidates when their *dedicated* argument is non-`nil`. The behavior of `set-window-buffer` (see Section 29.11 [Buffers and Windows], page 722) with respect to dedicated windows is slightly different, see below.

Functions supposed to remove a buffer from a window or a window from a frame can behave specially when a window they operate on is dedicated. We will distinguish four basic cases, namely where (1) the window is not the only window on its frame, (2) the window is the only window on its frame but there are other frames on the same terminal left, (3) the window is the only window on the only frame on the same terminal, and (4) the dedication's value is `side` (see Section 29.17.1 [Displaying Buffers in Side Windows], page 754).

In particular, `delete-windows-on` (see Section 29.8 [Deleting Windows], page 713) handles case (2) by deleting the associated frame and cases (3) and (4) by showing another buffer in that frame's only window. The function `replace-buffer-in-windows` (see Section 29.11 [Buffers and Windows], page 722) which is called when a buffer gets killed, deletes the window in case (1) and behaves like `delete-windows-on` otherwise.

When `bury-buffer` (see Section 28.8 [Buffer List], page 683) operates on the selected window (which shows the buffer that shall be buried), it handles case (2) by calling `frame-auto-hide-function` (see Section 29.16 [Quitting Windows], page 752) to deal with the selected frame. The other two cases are handled as with `replace-buffer-in-windows`.

`window-dedicated-p` **&optional** *window* [Function]
: This function returns non-`nil` if *window* is dedicated to its buffer and `nil` otherwise. More precisely, the return value is the value assigned by the last call of `set-window-dedicated-p` for *window*, or `nil` if that function was never called with *window* as its argument. The default for *window* is the selected window.

`set-window-dedicated-p` *window flag* [Function]
: This function marks *window* as dedicated to its buffer if *flag* is non-`nil`, and non-dedicated otherwise.

  As a special case, if *flag* is `t`, *window* becomes *strongly* dedicated to its buffer. `set-window-buffer` signals an error when the window it acts upon is strongly dedicated to its buffer and does not already display the buffer it is asked to display. Other functions do not treat `t` differently from any non-`nil` value.

You can also tell `display-buffer` to mark a window it creates as dedicated to its buffer by providing a suitable `dedicated` action alist entry (see Section 29.13.3 [Buffer Display Action Alists], page 733).

29.16 Quitting Windows

After a command uses `display-buffer` to put a buffer on the screen, the user may decide to hide it and return to the previous configuration of the Emacs display. We call that *quitting the window*. The way to do this is to call `quit-window` while the window used by `display-buffer` is the selected window.

The right way to restore the previous configuration of the display depends on what was done to the window where the buffer now appears. It might be right to delete that window, or delete its frame, or just display another buffer in that window. One complication is that the user may have changed the window configuration since the act of displaying that buffer, and it would be undesirable to undo the user's explicitly requested changes.

To enable `quit-window` to do the right thing, `display-buffer` saves information about what it did in the window's `quit-restore` parameter (see Section 29.27 [Window Parameters], page 779).

`quit-window` **&optional** *kill window* [Command]
: This command quits *window* and buries its buffer. The argument *window* must be a live window and defaults to the selected one. With prefix argument *kill* non-`nil`, it kills the buffer instead of burying it.

  The function `quit-window` first runs `quit-window-hook`. Then it calls the function `quit-restore-window`, described next, which does the hard work.

You can get more control by calling `quit-restore-window` instead.

`quit-restore-window` **&optional** *window bury-or-kill* [Function]
: This function handles *window* and its buffer after quitting. The optional argument *window* must be a live window and defaults to the selected one. The function takes account of the *window*'s `quit-restore` parameter.

  The optional argument *bury-or-kill* specifies how to deal with *window*'s buffer. The following values are meaningful:

  `nil` This means to not deal with the buffer in any particular way. As a consequence, if *window* is not deleted, invoking `switch-to-prev-buffer` will usually show the buffer again.

  `append` This means that if *window* is not deleted, its buffer is moved to the end of *window*'s list of previous buffers (see Section 29.14 [Window History], page 748), so it's less likely that future invocations of `switch-to-prev-buffer` will switch to it. Also, it moves the buffer to the end of the frame's buffer list (see Section 28.8 [Buffer List], page 683).

  `bury` This means that if *window* is not deleted, its buffer is removed from *window*'s list of previous buffers. Also, it moves the buffer to the end of the frame's buffer list. This is the most reliable way to prevent `switch-to-prev-buffer` from switching to this buffer again, short of killing the buffer.

  `kill` This means to kill *window*'s buffer.

The argument *bury-or-kill* also specifies what to do with *window*'s frame when *window* should be deleted, if it is the only window on its frame, and there are other frames on that frame's terminal. If *bury-or-kill* equals `kill`, it means to delete the frame. Otherwise, the fate of the frame is determined by calling `frame-auto-hide-function` (see below) with that frame as sole argument.

This function always sets *window*'s `quit-restore` parameter to `nil` unless it deletes the window.

The window *window*'s `quit-restore` parameter (see Section 29.27 [Window Parameters], page 779) should be `nil` or a list of four elements:

```
(method obuffer owindow this-buffer)
```

The first element, *method*, is one of the four symbols `window`, `frame`, `same` and `other`. `frame` and `window` control how to delete *window*, while `same` and `other` control displaying some other buffer in it.

Specifically, `window` means that the window has been specially created by `display-buffer`; `frame` means that a separate frame has been created; `same`, that the window has only ever displayed this buffer; `other`, that the window showed another buffer before.

The second element, *obuffer*, is either one of the symbols `window` or `frame`, or a list of the form

```
(prev-buffer prev-window-start prev-window-point height)
```

which says which buffer was shown in *window* before, that buffer's window start (see Section 29.20 [Window Start and End], page 762) and window point (see Section 29.19 [Window Point], page 761) positions at that time, and *window*'s height at that time. If *prev-buffer* is still live when quitting *window*, quitting the window may reuse *window* to display *prev-buffer*.

The third element, *owindow*, is the window that was selected just before the displaying was done. If quitting deletes *window*, it tries to select *owindow*.

The fourth element, *this-buffer*, is the buffer whose displaying set the `quit-restore` parameter. Quitting *window* may delete that window only if it still shows that buffer.

Quitting *window* tries to delete it if and only if (1) *method* is either `window` or `frame`, (2) the window has no history of previously-displayed buffers and (3) *this-buffer* equals the buffer currently displayed in *window*. If *window* is part of an atomic window (see Section 29.18 [Atomic Windows], page 759), quitting will try to delete the root of that atomic window instead. In either case, it tries to avoid signaling an error when *window* cannot be deleted.

If *obuffer* is a list, and *prev-buffer* is still live, quitting displays *prev-buffer* in *window* according to the rest of the elements of *obuffer*. This includes resizing the window to *height* if it was temporarily resized to display *this-buffer*.

Otherwise, if *window* was previously used for displaying other buffers (see Section 29.14 [Window History], page 748), the most recent buffer in that history will be displayed.

The following option specifies a function to do the right thing with a frame containing one window when quitting that window.

`frame-auto-hide-function` [User Option]

The function specified by this option is called to automatically hide frames. This function is called with one argument—a frame.

The function specified here is called by `bury-buffer` (see Section 28.8 [Buffer List], page 683) when the selected window is dedicated and shows the buffer to bury. It is also called by `quit-restore-window` (see above) when the frame of the window to quit has been specially created for displaying that window's buffer and the buffer is not killed.

The default is to call `iconify-frame` (see Section 30.11 [Visibility of Frames], page 831). Alternatively, you may specify either `delete-frame` (see Section 30.7 [Deleting Frames], page 824) to remove the frame from its display, `make-frame-invisible` to make the frame invisible, `ignore` to leave the frame unchanged, or any other function that can take a frame as its sole argument.

Note that the function specified by this option is called only if the specified frame contains just one live window and there is at least one other frame on the same terminal.

For a particular frame, the value specified here may be overridden by that frame's `auto-hide-function` frame parameter (see Section 30.4.3.6 [Frame Interaction Parameters], page 814).

29.17 Side Windows

Side windows are special windows positioned at any of the four sides of a frame's root window (see Section 29.2 [Windows and Frames], page 694). In practice, this means that the area of the frame's root window is subdivided into a main window and a number of side windows surrounding that main window. The main window is either a "normal" live window or specifies the area containing all the normal windows.

In their most simple form of use, side windows allow to display specific buffers always in the same area of a frame. Hence they can be regarded as a generalization of the concept provided by `display-buffer-at-bottom` (see Section 29.13.2 [Buffer Display Action Functions], page 728) to the remaining sides of a frame. With suitable customizations, however, side windows can be also used to provide frame layouts similar to those found in so-called integrated development environments (IDEs).

29.17.1 Displaying Buffers in Side Windows

The following action function for `display-buffer` (see Section 29.13.2 [Buffer Display Action Functions], page 728) creates or reuses a side window for displaying the specified buffer.

`display-buffer-in-side-window` *buffer alist* [Function]

This function displays *buffer* in a side window of the selected frame. It returns the window used for displaying *buffer*, `nil` if no such window can be found or created.

alist is an association list of symbols and values as for `display-buffer`. The following symbols in *alist* are special for this function:

`side` Denotes the side of the frame where the window shall be located. Valid values are `left`, `top`, `right` and `bottom`. If unspecified, the window is located at the bottom of the frame.

`slot` Denotes a slot at the specified side where to locate the window. A value of zero means to preferably position the window in the middle of the specified side. A negative value means to use a slot preceding (that is, above or on the left of) the middle slot. A positive value means to use a slot following (that is, below or on the right of) the middle slot. Hence, all windows on a specific side are ordered by their `slot` value. If unspecified, the window is located in the middle of the specified side.

`dedicated`
The dedicated flag (see Section 29.15 [Dedicated Windows], page 751) has a slightly different meaning for side windows. When a side window is created, that flag is set to the value `side` to prevent `display-buffer` to use the window in other action functions. Its value persists across invocations of `quit-window`, `kill-buffer`, `previous-buffer` and `next-buffer`.

In particular, these commands will refrain from showing, in a side window, buffers that have not been displayed in that window before. They will also refrain from having a normal, non-side window show a buffer that has been already displayed in a side window. A notable exception to the latter rule occurs when an application, after displaying a buffer, resets that buffer's local variables. To override these rules and always delete a side window with `quit-window` or `kill-buffer`, and eventually prevent the use of `previous-buffer` and `next-buffer`, set this value to `t` or specify a value via `display-buffer-mark-dedicated`.

If you specify the same slot on the same side for two or more different buffers, the buffer displayed last is shown in the corresponding window. Hence, slots can be used for sharing the same side window between buffers.

This function installs the `window-side` and `window-slot` parameters (see Section 29.27 [Window Parameters], page 779) and makes them persistent. It does not install any other window parameters unless they have been explicitly provided via a `window-parameters` entry in *alist*.

By default, side windows cannot be split via `split-window` (see Section 29.7 [Splitting Windows], page 710). Also, a side window is not reused or split by any buffer display action (see Section 29.13.2 [Buffer Display Action Functions], page 728) unless it is explicitly specified as target of that action. Note also that `delete-other-windows` cannot make a side window the only window on its frame (see Section 29.8 [Deleting Windows], page 713).

29.17.2 Side Window Options and Functions

The following options provide additional control over the placement of side windows.

`window-sides-vertical` [User Option]
If non-`nil`, the side windows on the left and right of a frame occupy the frame's full height. Otherwise, the side windows on the top and bottom of the frame occupy the frame's full width.

`window-sides-slots` [User Option]
This option specifies the maximum number of side windows on each side of a frame. The value is a list of four elements specifying the number of side window slots on (in

this order) the left, top, right and bottom of each frame. If an element is a number, it means to display at most that many windows on the corresponding side. If an element is `nil`, it means there's no bound on the number of slots on that side.

If any of the specified values is zero, no window can be created on the corresponding side. `display-buffer-in-side-window` will not signal an error in that case, but will return `nil`. If a specified value just forbids the creation of an additional side window, the most suitable window on that side is reused and may have its `window-slot` parameter changed accordingly.

`window-sides-reversed` [User Option]

This option specifies whether top/bottom side windows should appear in reverse order. When this is `nil`, side windows on the top and bottom of a frame are always drawn from left to right with increasing slot values. When this is `t`, the drawing order is reversed and side windows on the top and bottom of a frame are drawn from right to left with increasing slot values.

When this is `bidi`, the drawing order is reversed if and only if the value of `bidi-paragraph-direction` (see Section 41.27 [Bidirectional Display], page 1255) is `right-to-left` in the buffer displayed in the window most recently selected within the main window area of this frame. Sometimes that window may be hard to find, so heuristics are used to avoid that the drawing order changes inadvertently when another window gets selected.

The layout of side windows on the left or right of a frame is not affected by the value of this variable.

When a frame has side windows, the following function returns the main window of that frame.

`window-main-window` **&optional** *frame* [Function]

This function returns the main window of the specified *frame*. The optional argument *frame* must be a live frame and defaults to the selected one.

If *frame* has no side windows, it returns *frame*'s root window. Otherwise, it returns either an internal non-side window such that all other non-side windows on *frame* descend from it, or the single live non-side window of *frame*. Note that the main window of a frame cannot be deleted via `delete-window`.

The following command is handy to toggle the appearance of all side windows on a specified frame.

`window-toggle-side-windows` **&optional** *frame* [Command]

This command toggles side windows on the specified *frame*. The optional argument *frame* must be a live frame and defaults to the selected one.

If *frame* has at least one side window, this command saves the state of *frame*'s root window in the *frame*'s `window-state` frame parameter and deletes all side windows on *frame* afterwards.

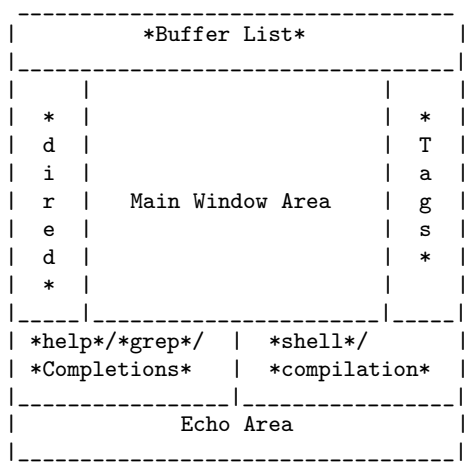
If *frame* has no side windows, but does have a `window-state` parameter, this command uses that parameter's value to restore the side windows on *frame* leaving *frame*'s main window alone.

An error is signaled if *frame* has no side windows and no saved state is found for it.

29.17.3 Frame Layouts with Side Windows

Side windows can be used to create more complex frame layouts like those provided by integrated development environments (IDEs). In such layouts, the area of the main window is where the normal editing activities take place. Side windows are not conceived for editing in the usual sense. Rather, they are supposed to display information complementary to the current editing activity, like lists of files, tags or buffers, help information, search or grep results or shell output.

The layout of such a frame might appear as follows:



The following example illustrates how window parameters (see Section 29.27 [Window Parameters], page 779) can be used with `display-buffer-in-side-window` (see Section 29.17.1 [Displaying Buffers in Side Windows], page 754) to set up code for producing the frame layout sketched above.

```
(defvar parameters
  '(window-parameters . ((no-other-window . t)
                         (no-delete-other-windows . t))))

(setq fit-window-to-buffer-horizontally t)
(setq window-resize-pixelwise t)

(setq
 display-buffer-alist
 `(("\\*Buffer List\\*" display-buffer-in-side-window
    (side . top) (slot . 0) (window-height . fit-window-to-buffer)
    (preserve-size . (nil . t)) ,parameters)
   ("\\*Tags List\\*" display-buffer-in-side-window
    (side . right) (slot . 0) (window-width . fit-window-to-buffer)
    (preserve-size . (t . nil)) ,parameters)
   ("\\*\\(?:help\\|grep\\|Completions\\)\\*"
    display-buffer-in-side-window
    (side . bottom) (slot . -1) (preserve-size . (nil . t))
    ,parameters)
   ("\\*\\(?:shell\\|compilation\\)\\*" display-buffer-in-side-window
    (side . bottom) (slot . 1) (preserve-size . (nil . t))
    ,parameters)))
```

This specifies `display-buffer-alist` entries (see Section 29.13.1 [Choosing Window], page 726) for buffers with fixed names. In particular, it asks for showing `*Buffer List*` with adjustable height at the top of the frame and `*Tags List*` with adjustable width on the frame's right. It also asks for having the `*help*`, `*grep*` and `*Completions*` buffers share a window on the bottom left side of the frame and the `*shell*` and `*compilation*` buffers appear in a window on the bottom right side of the frame.

Note that the option `fit-window-to-buffer-horizontally` must have a non-`nil` value in order to allow horizontal adjustment of windows. Entries are also added that ask for preserving the height of side windows at the top and bottom of the frame and the width of side windows at the left or right of the frame. To assure that side windows retain their respective sizes when maximizing the frame, the variable `window-resize-pixelwise` is set to a non-`nil` value. See Section 29.5 [Resizing Windows], page 705.

The last form also makes sure that none of the created side windows are accessible via *C-x o* by installing the `no-other-window` parameter for each of these windows. In addition, it makes sure that side windows are not deleted via *C-x 1* by installing the `no-delete-other-windows` parameter for each of these windows.

Since `dired` buffers have no fixed names, we use a special function `dired-default-directory-on-left` in order to display a lean directory buffer on the left side of the frame.

```
(defun dired-default-directory-on-left ()
  "Display `default-directory' in side window on left, hiding details."
  (interactive)
  (let ((buffer (dired-noselect default-directory)))
    (with-current-buffer buffer (dired-hide-details-mode t))
    (display-buffer-in-side-window
     buffer `((side . left) (slot . 0)
              (window-width . fit-window-to-buffer)
              (preserve-size . (t . nil)) ,parameters))))
```

Evaluating the preceding forms and typing, in any order, *M-x list-buffers*, *C-h f*, *M-x shell*, *M-x list-tags*, and *M-x dired-default-directory-on-left* should now reproduce the frame layout sketched above.

29.18 Atomic Windows

Atomic windows are rectangular compositions of at least two live windows. They have the following distinctive characteristics:

- The function `split-window` (see Section 29.7 [Splitting Windows], page 710), when applied to a constituent of an atomic window, will try to create the new window outside of the atomic window.
- The function `delete-window` (see Section 29.8 [Deleting Windows], page 713), when applied to a constituent of an atomic window, will try to delete the entire atomic window instead.
- The function `delete-other-windows` (see Section 29.8 [Deleting Windows], page 713), when applied to a constituent of an atomic window, will try to make the atomic window fill its frame or main window (see Section 29.17 [Side Windows], page 754).

This means that the basic groups of functions that alter the window structure treat an atomic window like a live one, thus preserving the internal structure of the atomic window.

Atomic windows are useful to construct and preserve window layouts that are meaningful only when all involved buffers are shown simultaneously in a specific manner, such as when showing differences between file revisions, or the same text in different languages or markups. They can also be used to permanently display information pertinent to a specific window in bars on that window's sides.

Atomic windows are implemented with the help of the reserved `window-atom` window parameter (see Section 29.27 [Window Parameters], page 779) and an internal window (see Section 29.1 [Basic Windows], page 692) called the root window of the atomic window. All windows that are part of the same atomic window have this root window as their common ancestor and are assigned a non-`nil` `window-atom` parameter.

The following function returns the root of the atomic window a specified window is part of:

`window-atom-root` **&optional** *window* [Function]

This functions returns the root of the atomic window *window* is a part of. The specified *window* must be a valid window and defaults to the selected one. It returns `nil` if *window* is not part of an atomic window.

The most simple approach to make a new atomic window is to take an existing internal window and apply the following function:

`window-make-atom` *window* [Function]
: This function converts *window* into an atomic window. The specified *window* must be an internal window. All this function does is to set the `window-atom` parameter of each descendant of *window* to `t`.

To create a new atomic window from an existing live window or to add a new window to an existing atomic window, the following buffer display action function (see Section 29.13.2 [Buffer Display Action Functions], page 728) can be used:

`display-buffer-in-atom-window` *buffer alist* [Function]
: This function tries to display *buffer* in a new window that will be combined with an existing window to form an atomic window. If the existing window is already part of an atomic window, it adds the new window to that atomic window.

  The specified *alist* is an association list of symbols and values. The following symbols have a special meaning:

  `window` The value of such an element specifies an existing window the new window shall be combined with. If it specifies an internal window, all children of that window become part of the atomic window too. If no window is specified, the new window becomes a sibling of the selected window. The `window-atom` parameter of the existing window is set to `main` provided that window is live and its `window-atom` parameter was not already set.

  `side` The value of such an element denotes the side of the existing window where the new window shall be located. Valid values are `below`, `right`, `above` and `left`. The default is `below`. The `window-atom` parameter of the new window is set to this value.

  The return value is the new window, `nil` when creating that window failed.

Note that the value of the `window-atom` parameter does not really matter as long as it is non-`nil`. The values assigned by `display-buffer-in-atom-window` just allow for easy retrieval of the original and the new window after that function has been applied. Note also that the `window-atom` parameter is the only window parameter assigned by `display-buffer-in-atom-window`. Further parameters have to be set by the application explicitly via a `window-parameters` entry in *alist*.

Atomic windows automatically cease to exist when one of their constituents gets deleted. To dissolve an atomic window manually, reset the `window-atom` parameter of its constituents—the root of the atomic window and all its descendants.

The following code snippet, when applied to a single-window frame, first splits the selected window and makes the selected and the new window constituents of an atomic window with their parent as root. It then displays the buffer `*Messages*` in a new window at the frame's bottom and makes that new window part of the atomic window just created.

```
(let ((window (split-window-right)))
  (window-make-atom (window-parent window))
  (display-buffer-in-atom-window
   (get-buffer-create "*Messages*")
   `((window . ,(window-parent window)) (window-height . 5))))
```

At this moment typing `C-x 2` in any window of that frame produces a new window at the bottom of the frame. Typing `C-x 3` instead will put the new window at the frame's right. In either case, typing now `C-x 1` in any window of the atomic window will remove the new window only. Typing `C-x 0` in any window of the atomic window will make that new window fill the frame.

29.19 Windows and Point

Each window has its own value of point (see Section 31.1 [Point], page 855), independent of the value of point in other windows displaying the same buffer. This makes it useful to have multiple windows showing one buffer.

- The window point is established when a window is first created; it is initialized from the buffer's point, or from the window point of another window opened on the buffer if such a window exists.
- Selecting a window sets the value of point in its buffer from the window's value of point. Conversely, deselecting a window sets the window's value of point from that of the buffer. Thus, when you switch between windows that display a given buffer, the point value for the selected window is in effect in the buffer, while the point values for the other windows are stored in those windows.
- As long as the selected window displays the current buffer, the window's point and the buffer's point always move together; they remain equal.

Emacs displays the cursor, by default as a rectangular block, in each window at the position of that window's point. When the user switches to another buffer in a window, Emacs moves that window's cursor to where point is in that buffer. If the exact position of point is hidden behind some display element, such as a display string or an image, Emacs displays the cursor immediately before or after that display element.

`window-point` **&optional** *window* [Function]

This function returns the current position of point in *window*. For a nonselected window, this is the value point would have (in that window's buffer) if that window were selected. The default for *window* is the selected window.

When *window* is the selected window, the value returned is the value of point in that window's buffer. Strictly speaking, it would be more correct to return the top-level value of point, outside of any `save-excursion` forms. But that value is hard to find.

`set-window-point` *window position* [Function]

This function positions point in *window* at position *position* in *window*'s buffer. It returns *position*.

If *window* is selected, this simply does `goto-char` in *window*'s buffer.

`window-point-insertion-type` [Variable]
This variable specifies the marker insertion type (see Section 32.5 [Marker Insertion Types], page 874) of `window-point`. The default is `nil`, so `window-point` will stay behind text inserted there.

29.20 The Window Start and End Positions

Each window maintains a marker used to keep track of a buffer position that specifies where in the buffer display should start. This position is called the *display-start* position of the window (or just the *start*). The character after this position is the one that appears at the upper left corner of the window. It is usually, but not inevitably, at the beginning of a text line.

After switching windows or buffers, and in some other cases, if the window start is in the middle of a line, Emacs adjusts the window start to the start of a line. This prevents certain operations from leaving the window start at a meaningless point within a line. This feature may interfere with testing some Lisp code by executing it using the commands of Lisp mode, because they trigger this readjustment. To test such code, put it into a command and bind the command to a key.

`window-start` **&optional** *window* [Function]
This function returns the display-start position of window *window*. If *window* is `nil`, the selected window is used.

When you create a window, or display a different buffer in it, the display-start position is set to a display-start position recently used for the same buffer, or to `point-min` if the buffer doesn't have any.

Redisplay updates the window-start position (if you have not specified it explicitly since the previous redisplay)—to make sure point appears on the screen. Nothing except redisplay automatically changes the window-start position; if you move point, do not expect the window-start position to change in response until after the next redisplay.

`window-group-start` **&optional** *window* [Function]
This function is like `window-start`, except that when *window* is a part of a group of windows (see [Window Group], page 700), `window-group-start` returns the start position of the entire group. This condition holds when the buffer local variable `window-group-start-function` is set to a function. In this case, `window-group-start` calls the function with the single argument *window*, then returns its result.

`window-end` **&optional** *window update* [Function]
This function returns the position where display of its buffer ends in *window*. The default for *window* is the selected window.

Simply changing the buffer text or moving point does not update the value that `window-end` returns. The value is updated only when Emacs redisplays and redisplay completes without being preempted.

If the last redisplay of *window* was preempted, and did not finish, Emacs does not know the position of the end of display in that window. In that case, this function returns `nil`.

If *update* is non-`nil`, `window-end` always returns an up-to-date value for where display ends, based on the current `window-start` value. If a previously saved value of that position is still valid, `window-end` returns that value; otherwise it computes the correct value by scanning the buffer text.

Even if *update* is non-`nil`, `window-end` does not attempt to scroll the display if point has moved off the screen, the way real redisplay would do. It does not alter the `window-start` value. In effect, it reports where the displayed text will end if scrolling is not required. Note that the position it returns might be only partially visible.

`window-group-end` **&optional** *window update* [Function]
This function is like `window-end`, except that when *window* is a part of a group of windows (see [Window Group], page 700), `window-group-end` returns the end position of the entire group. This condition holds when the buffer local variable `window-group-end-function` is set to a function. In this case, `window-group-end` calls the function with the two arguments *window* and *update*, then returns its result. The argument *update* has the same meaning as in `window-end`.

`set-window-start` *window position* **&optional** *noforce* [Function]
This function sets the display-start position of *window* to *position* in *window*'s buffer. It returns *position*.

The display routines insist that the position of point be visible when a buffer is displayed. Normally, they select the display-start position according to their internal logic (and scroll the window if necessary) to make point visible. However, if you specify the start position with this function using `nil` for *noforce*, it means you want display to start at *position* even if that would put the location of point off the screen. If this does place point off screen, the display routines attempt to move point to the left margin on the middle line in the window.

For example, if point is 1 and you set the start of the window to 37, the start of the next line, point will be above the top of the window. The display routines will automatically move point if it is still 1 when redisplay occurs. Here is an example:

```
;; Here is what 'foo' looks like before executing
;;    the set-window-start expression.

---------- Buffer: foo ----------
*This is the contents of buffer foo.
2
3
4
5
6
---------- Buffer: foo ----------
```

```
(set-window-start
 (selected-window)
 (save-excursion
   (goto-char 1)
   (forward-line 1)
   (point)))
⇒ 37

;; Here is what 'foo' looks like after executing
;;   the set-window-start expression.
---------- Buffer: foo ----------
2
3
⋆4
5
6
---------- Buffer: foo ----------
```

If the attempt to make point visible (i.e., in a fully-visible screen line) fails, the display routines will disregard the requested window-start position and compute a new one anyway. Thus, for reliable results Lisp programs that call this function should always move point to be inside the window whose display starts at *position*.

If *noforce* is non-`nil`, and *position* would place point off screen at the next redisplay, then redisplay computes a new window-start position that works well with point, and thus *position* is not used.

`set-window-group-start` *window position* **&optional** *noforce* [Function]

This function is like `set-window-start`, except that when *window* is a part of a group of windows (see [Window Group], page 700), `set-window-group-start` sets the start position of the entire group. This condition holds when the buffer local variable `set-window-group-start-function` is set to a function. In this case, `set-window-group-start` calls the function with the three arguments *window*, *position*, and *noforce*, then returns its result. The arguments *position* and *noforce* in this function have the same meaning as in `set-window-start`.

`pos-visible-in-window-p` **&optional** *position window partially* [Function]

This function returns non-`nil` if *position* is within the range of text currently visible on the screen in *window*. It returns `nil` if *position* is scrolled vertically out of view. Locations that are partially obscured are not considered visible unless *partially* is non-`nil`. The argument *position* defaults to the current position of point in *window*; *window* defaults to the selected window. If *position* is `t`, that means to check either the first visible position of the last screen line in *window*, or the end-of-buffer position, whichever comes first.

This function considers only vertical scrolling. If *position* is out of view only because *window* has been scrolled horizontally, `pos-visible-in-window-p` returns non-`nil` anyway. See Section 29.23 [Horizontal Scrolling], page 770.

If *position* is visible, `pos-visible-in-window-p` returns `t` if *partially* is `nil`; if *partially* is non-`nil`, and the character following *position* is fully visible, it returns a list

of the form `(x y)`, where *x* and *y* are the pixel coordinates relative to the top left corner of the window; otherwise it returns an extended list of the form `(x y rtop rbot rowh vpos)`, where *rtop* and *rbot* specify the number of off-window pixels at the top and bottom of the row at *position*, *rowh* specifies the visible height of that row, and *vpos* specifies the vertical position (zero-based row number) of that row.

Here is an example:

```
;; If point is off the screen now, recenter it now.
(or (pos-visible-in-window-p
     (point) (selected-window))
    (recenter 0))
```

`pos-visible-in-window-group-p` **&optional** *position window partially* [Function]

This function is like `pos-visible-in-window-p`, except that when *window* is a part of a group of windows (see [Window Group], page 700), `pos-visible-in-window-group-p` tests the visibility of *pos* in the entire group, not just in the single *window*. This condition holds when the buffer local variable `pos-visible-in-window-group-p-function` is set to a function. In this case `pos-visible-in-window-group-p` calls the function with the three arguments *position*, *window*, and *partially*, then returns its result. The arguments *position* and *partially* have the same meaning as in `pos-visible-in-window-p`.

`window-line-height` **&optional** *line window* [Function]

This function returns the height of text line *line* in *window*. If *line* is one of `header-line` or `mode-line`, `window-line-height` returns information about the corresponding line of the window. Otherwise, *line* is a text line number starting from 0. A negative number counts from the end of the window. The default for *line* is the current line in *window*; the default for *window* is the selected window.

If the display is not up to date, `window-line-height` returns `nil`. In that case, `pos-visible-in-window-p` may be used to obtain related information.

If there is no line corresponding to the specified *line*, `window-line-height` returns `nil`. Otherwise, it returns a list `(height vpos ypos offbot)`, where *height* is the height in pixels of the visible part of the line, *vpos* and *ypos* are the vertical position in lines and pixels of the line relative to the top of the first text line, and *offbot* is the number of off-window pixels at the bottom of the text line. If there are off-window pixels at the top of the (first) text line, *ypos* is negative.

29.21 Textual Scrolling

Textual scrolling means moving the text up or down through a window. It works by changing the window's display-start location. It may also change the value of `window-point` to keep point on the screen (see Section 29.19 [Window Point], page 761).

The basic textual scrolling functions are `scroll-up` (which scrolls forward) and `scroll-down` (which scrolls backward). In these function names, "up" and "down" refer to the direction of motion of the buffer text relative to the window. Imagine that the text is written on a long roll of paper and that the scrolling commands move the paper up and

down. Thus, if you are looking at the middle of a buffer and repeatedly call `scroll-down`, you will eventually see the beginning of the buffer.

Unfortunately, this sometimes causes confusion, because some people tend to think in terms of the opposite convention: they imagine the window moving over text that remains in place, so that "down" commands take you to the end of the buffer. This convention is consistent with fact that such a command is bound to a key named `PageDown` on modern keyboards.

Textual scrolling functions (aside from `scroll-other-window`) have unpredictable results if the current buffer is not the one displayed in the selected window. See Section 28.2 [Current Buffer], page 673.

If the window contains a row taller than the height of the window (for example in the presence of a large image), the scroll functions will adjust the window's vertical scroll position to scroll the partially visible row. Lisp callers can disable this feature by binding the variable `auto-window-vscroll` to `nil` (see Section 29.22 [Vertical Scrolling], page 769).

`scroll-up` **&optional** *count* [Command]

This function scrolls forward by *count* lines in the selected window.

If *count* is negative, it scrolls backward instead. If *count* is `nil` (or omitted), the distance scrolled is `next-screen-context-lines` lines less than the height of the window's body.

If the selected window cannot be scrolled any further, this function signals an error. Otherwise, it returns `nil`.

`scroll-down` **&optional** *count* [Command]

This function scrolls backward by *count* lines in the selected window.

If *count* is negative, it scrolls forward instead. In other respects, it behaves the same way as `scroll-up` does.

`scroll-up-command` **&optional** *count* [Command]

This behaves like `scroll-up`, except that if the selected window cannot be scrolled any further and the value of the variable `scroll-error-top-bottom` is `t`, it tries to move to the end of the buffer instead. If point is already there, it signals an error.

`scroll-down-command` **&optional** *count* [Command]

This behaves like `scroll-down`, except that if the selected window cannot be scrolled any further and the value of the variable `scroll-error-top-bottom` is `t`, it tries to move to the beginning of the buffer instead. If point is already there, it signals an error.

`scroll-other-window` **&optional** *count* [Command]

This function scrolls the text in another window upward *count* lines. Negative values of *count*, or `nil`, are handled as in `scroll-up`.

You can specify which buffer to scroll by setting the variable `other-window-scroll-buffer` to a buffer. If that buffer isn't already displayed, `scroll-other-window` displays it in some window.

When the selected window is the minibuffer, the next window is normally the leftmost one immediately above it. You can specify a different window to scroll, when

the minibuffer is selected, by setting the variable `minibuffer-scroll-window`. This variable has no effect when any other window is selected. When it is non-`nil` and the minibuffer is selected, it takes precedence over `other-window-scroll-buffer`. See [Definition of minibuffer-scroll-window], page 423.

`scroll-other-window-down` **&optional** *count* [Command]
This function scrolls the text in another window downward *count* lines. Negative values of *count*, or `nil`, are handled as in `scroll-down`. In other respects, it behaves the same way as `scroll-other-window` does.

`other-window-scroll-buffer` [Variable]
If this variable is non-`nil`, it tells `scroll-other-window` which buffer's window to scroll.

`scroll-margin` [User Option]
This option specifies the size of the scroll margin—a minimum number of lines between point and the top or bottom of a window. Whenever point gets within this many lines of the top or bottom of the window, redisplay scrolls the text automatically (if possible) to move point out of the margin, closer to the center of the window.

`maximum-scroll-margin` [User Option]
This variable limits the effective value of `scroll-margin` to a fraction of the current window line height. For example, if the current window has 20 lines and `maximum-scroll-margin` is 0.1, then the scroll margins will never be larger than 2 lines, no matter how big `scroll-margin` is.

`maximum-scroll-margin` itself has a maximum value of 0.5, which allows setting margins large to keep the cursor at the middle line of the window (or two middle lines if the window has an even number of lines). If it's set to a larger value (or any value other than a float between 0.0 and 0.5) then the default value of 0.25 will be used instead.

`scroll-conservatively` [User Option]
This variable controls how scrolling is done automatically when point moves off the screen (or into the scroll margin). If the value is a positive integer *n*, then redisplay scrolls the text up to *n* lines in either direction, if that will bring point back into proper view. This behavior is called *conservative scrolling*. Otherwise, scrolling happens in the usual way, under the control of other variables such as `scroll-up-aggressively` and `scroll-down-aggressively`.

The default value is zero, which means that conservative scrolling never happens.

`scroll-down-aggressively` [User Option]
The value of this variable should be either `nil` or a fraction *f* between 0 and 1. If it is a fraction, that specifies where on the screen to put point when scrolling down. More precisely, when a window scrolls down because point is above the window start, the new start position is chosen to put point *f* part of the window height from the top. The larger *f*, the more aggressive the scrolling.

A value of `nil` is equivalent to .5, since its effect is to center point. This variable automatically becomes buffer-local when set in any fashion.

`scroll-up-aggressively` [User Option]
Likewise, for scrolling up. The value, *f*, specifies how far point should be placed from the bottom of the window; thus, as with `scroll-down-aggressively`, a larger value scrolls more aggressively.

`scroll-step` [User Option]
This variable is an older variant of `scroll-conservatively`. The difference is that if its value is *n*, that permits scrolling only by precisely *n* lines, not a smaller number. This feature does not work with `scroll-margin`. The default value is zero.

`scroll-preserve-screen-position` [User Option]
If this option is `t`, whenever a scrolling command moves point off-window, Emacs tries to adjust point to keep the cursor at its old vertical position in the window, rather than the window edge.

If the value is non-`nil` and not `t`, Emacs adjusts point to keep the cursor at the same vertical position, even if the scrolling command didn't move point off-window.

This option affects all scroll commands that have a non-`nil` `scroll-command` symbol property.

`next-screen-context-lines` [User Option]
The value of this variable is the number of lines of continuity to retain when scrolling by full screens. For example, `scroll-up` with an argument of `nil` scrolls so that this many lines at the bottom of the window appear instead at the top. The default value is 2.

`scroll-error-top-bottom` [User Option]
If this option is `nil` (the default), `scroll-up-command` and `scroll-down-command` simply signal an error when no more scrolling is possible.

If the value is `t`, these commands instead move point to the beginning or end of the buffer (depending on scrolling direction); only if point is already on that position do they signal an error.

`recenter` **&optional** *count redisplay* [Command]
This function scrolls the text in the selected window so that point is displayed at a specified vertical position within the window. It does not move point with respect to the text.

If *count* is a non-negative number, that puts the line containing point *count* lines down from the top of the window. If *count* is a negative number, then it counts upward from the bottom of the window, so that −1 stands for the last usable line in the window.

If *count* is `nil` (or a non-`nil` list), `recenter` puts the line containing point in the middle of the window. If *count* is `nil` and *redisplay* is non-`nil`, this function may redraw the frame, according to the value of `recenter-redisplay`. Thus, omitting the second argument can be used to countermand the effect of `recenter-redisplay` being non-`nil`. Interactive calls pass non-`nil` for *redisplay*.

When `recenter` is called interactively, *count* is the raw prefix argument. Thus, typing `C-u` as the prefix sets the *count* to a non-`nil` list, while typing `C-u 4` sets *count* to 4, which positions the current line four lines from the top.

With an argument of zero, `recenter` positions the current line at the top of the window. The command `recenter-top-bottom` offers a more convenient way to achieve this.

`recenter-window-group` **&optional** *count* [Function]
This function is like `recenter`, except that when the selected window is part of a group of windows (see [Window Group], page 700), `recenter-window-group` scrolls the entire group. This condition holds when the buffer local variable `recenter-window-group-function` is set to a function. In this case, `recenter-window-group` calls the function with the argument *count*, then returns its result. The argument *count* has the same meaning as in `recenter`, but with respect to the entire window group.

`recenter-redisplay` [User Option]
If this variable is non-`nil`, calling `recenter` with a `nil` *count* argument and non-`nil` *redisplay* argument redraws the frame. The default value is `tty`, which means only redraw the frame if it is a tty frame.

`recenter-top-bottom` **&optional** *count* [Command]
This command, which is the default binding for `C-l`, acts like `recenter`, except if called with no argument. In that case, successive calls place point according to the cycling order defined by the variable `recenter-positions`.

`recenter-positions` [User Option]
This variable controls how `recenter-top-bottom` behaves when called with no argument. The default value is `(middle top bottom)`, which means that successive calls of `recenter-top-bottom` with no argument cycle between placing point at the middle, top, and bottom of the window.

29.22 Vertical Fractional Scrolling

Vertical fractional scrolling means shifting text in a window up or down by a specified multiple or fraction of a line. Emacs uses it, for example, on images and screen lines which are taller than the window. Each window has a *vertical scroll position*, which is a number, never less than zero. It specifies how far to raise the contents of the window when displaying them. Raising the window contents generally makes all or part of some lines disappear off the top, and all or part of some other lines appear at the bottom. The usual value is zero.

The vertical scroll position is measured in units of the normal line height, which is the height of the default font. Thus, if the value is .5, that means the window contents will be scrolled up half the normal line height. If it is 3.3, that means the window contents are scrolled up somewhat over three times the normal line height.

What fraction of a line the vertical scrolling covers, or how many lines, depends on what the lines contain. A value of .5 could scroll a line whose height is very short off the screen, while a value of 3.3 could scroll just part of the way through a tall line or an image.

`window-vscroll` **&optional** *window pixels-p* [Function]
This function returns the current vertical scroll position of *window*. The default for *window* is the selected window. If *pixels-p* is non-`nil`, the return value is measured in pixels, rather than in units of the normal line height.

```
(window-vscroll)
     ⇒ 0
```

`set-window-vscroll` *window lines* **&optional** *pixels-p preserve-vscroll-p* [Function]

This function sets *window*'s vertical scroll position to *lines*. If *window* is `nil`, the selected window is used. The argument *lines* should be zero or positive; if not, it is taken as zero.

The actual vertical scroll position must always correspond to an integral number of pixels, so the value you specify is rounded accordingly.

The return value is the result of this rounding.

```
(set-window-vscroll (selected-window) 1.2)
     ⇒ 1.13
```

If *pixels-p* is non-`nil`, *lines* specifies a number of pixels. In this case, the return value is *lines*.

Normally, the vscroll does not take effect on windows that aren't the `minibuffer-scroll-window` or the selected window when the mini-window is resized (see Section 21.11 [Minibuffer Windows], page 419). This "frozen" behavior is disabled when the *preserve-vscroll-p* parameter is non-`nil`, which means to set the vscroll as usual.

`auto-window-vscroll` [Variable]

If this variable is non-`nil`, the `line-move`, `scroll-up`, and `scroll-down` functions will automatically modify the vertical scroll position to scroll through display rows that are taller than the height of the window, for example in the presence of large images.

29.23 Horizontal Scrolling

Horizontal scrolling means shifting the image in the window left or right by a specified multiple of the normal character width. Each window has a *horizontal scroll position*, which is a number, never less than zero. It specifies how far to shift the contents left. Shifting the window contents left generally makes all or part of some characters disappear off the left, and all or part of some other characters appear at the right. The usual value is zero.

The horizontal scroll position is measured in units of the normal character width, which is the width of space in the default font. Thus, if the value is 5, that means the window contents are scrolled left by 5 times the normal character width. How many characters actually disappear off to the left depends on their width, and could vary from line to line.

Because we read from side to side in the inner loop, and from top to bottom in the outer loop, the effect of horizontal scrolling is not like that of textual or vertical scrolling. Textual scrolling involves selection of a portion of text to display, and vertical scrolling moves the window contents contiguously; but horizontal scrolling causes part of *each line* to go off screen.

Usually, no horizontal scrolling is in effect; then the leftmost column is at the left edge of the window. In this state, scrolling to the right is meaningless, since there is no data to the left of the edge to be revealed by it; so this is not allowed. Scrolling to the left is allowed; it scrolls the first columns of text off the edge of the window and can reveal additional columns on the right that were truncated before. Once a window has a nonzero amount of leftward

horizontal scrolling, you can scroll it back to the right, but only so far as to reduce the net horizontal scroll to zero. There is no limit to how far left you can scroll, but eventually all the text will disappear off the left edge.

If `auto-hscroll-mode` is set, redisplay automatically alters the horizontal scrolling of a window as necessary to ensure that point is always visible. However, you can still set the horizontal scrolling value explicitly. The value you specify serves as a lower bound for automatic scrolling, i.e., automatic scrolling will not scroll a window to a column less than the specified one.

The default value of `auto-hscroll-mode` is `t`; setting it to `current-line` activates a variant of automatic horizontal scrolling whereby only the line showing the cursor is horizontally scrolled to make point visible, the rest of the window is left either unscrolled, or at the minimum scroll amount set by `scroll-left` and `scroll-right`, see below.

`scroll-left` **&optional** *count set-minimum* [Command]

This function scrolls the selected window *count* columns to the left (or to the right if *count* is negative). The default for *count* is the window width, minus 2.

The return value is the total amount of leftward horizontal scrolling in effect after the change—just like the value returned by `window-hscroll` (below).

Note that text in paragraphs whose base direction is right-to-left (see Section 41.27 [Bidirectional Display], page 1255) moves in the opposite direction: e.g., it moves to the right when `scroll-left` is invoked with a positive value of *count*.

Once you scroll a window as far right as it can go, back to its normal position where the total leftward scrolling is zero, attempts to scroll any farther right have no effect.

If *set-minimum* is non-`nil`, the new scroll amount becomes the lower bound for automatic scrolling; that is, automatic scrolling will not scroll a window to a column less than the value returned by this function. Interactive calls pass non-`nil` for *set-minimum*.

`scroll-right` **&optional** *count set-minimum* [Command]

This function scrolls the selected window *count* columns to the right (or to the left if *count* is negative). The default for *count* is the window width, minus 2. Aside from the direction of scrolling, this works just like `scroll-left`.

`window-hscroll` **&optional** *window* [Function]

This function returns the total leftward horizontal scrolling of *window*—the number of columns by which the text in *window* is scrolled left past the left margin. (In right-to-left paragraphs, the value is the total amount of the rightward scrolling instead.) The default for *window* is the selected window.

The return value is never negative. It is zero when no horizontal scrolling has been done in *window* (which is usually the case).

```
(window-hscroll)
     ⇒ 0
(scroll-left 5)
     ⇒ 5
(window-hscroll)
     ⇒ 5
```

`set-window-hscroll` *window columns* [Function]

This function sets horizontal scrolling of *window*. The value of *columns* specifies the amount of scrolling, in terms of columns from the left margin (right margin in right-to-left paragraphs). The argument *columns* should be zero or positive; if not, it is taken as zero. Fractional values of *columns* are not supported at present.

Note that `set-window-hscroll` may appear not to work if you test it by evaluating a call with *M-:* in a simple way. What happens is that the function sets the horizontal scroll value and returns, but then redisplay adjusts the horizontal scrolling to make point visible, and this overrides what the function did. You can observe the function's effect if you call it while point is sufficiently far from the left margin that it will remain visible.

The value returned is *columns*.

```
(set-window-hscroll (selected-window) 10)
     ⇒ 10
```

Here is how you can determine whether a given position *position* is off the screen due to horizontal scrolling:

```
(defun hscroll-on-screen (window position)
  (save-excursion
    (goto-char position)
    (and
     (>= (- (current-column) (window-hscroll window)) 0)
     (< (- (current-column) (window-hscroll window))
        (window-width window)))))
```

29.24 Coordinates and Windows

This section describes functions that report positions of and within a window. Most of these functions report positions relative to an origin at the native position of the window's frame (see Section 30.3 [Frame Geometry], page 793). Some functions report positions relative to the origin of the display of the window's frame. In any case, the origin has the coordinates (0, 0) and X and Y coordinates increase rightward and downward respectively.

For the following functions, X and Y coordinates are reported in integer character units, i.e., numbers of lines and columns respectively. On a graphical display, each "line" and "column" corresponds to the height and width of the default character specified by the frame's default font (see Section 30.3.2 [Frame Font], page 799).

`window-edges` **&optional** *window body absolute pixelwise* [Function]

This function returns a list of the edge coordinates of *window*. If *window* is omitted or `nil`, it defaults to the selected window.

The return value has the form `(left top right bottom)`. These list elements are, respectively, the X coordinate of the leftmost column occupied by the window, the Y coordinate of the topmost row, the X coordinate one column to the right of the rightmost column, and the Y coordinate one row down from the bottommost row.

Note that these are the actual outer edges of the window, including any of its decorations. On a text terminal, if the window has a neighbor on its right, its right edge includes the separator line between the window and its neighbor.

If the optional argument *body* is `nil`, this means to return the edges corresponding to the total size of *window*. *body* non-`nil` means to return the edges of *window*'s body. If *body* is non-`nil`, *window* must specify a live window.

If the optional argument *absolute* is `nil`, this means to return edges relative to the native position of *window*'s frame. *absolute* non-`nil` means to return coordinates relative to the origin (0, 0) of *window*'s display. On non-graphical systems this argument has no effect.

If the optional argument *pixelwise* is `nil`, this means to return the coordinates in terms of the default character width and height of *window*'s frame (see Section 30.3.2 [Frame Font], page 799), rounded if necessary. *pixelwise* non-`nil` means to return the coordinates in pixels. Note that the pixel specified by *right* and *bottom* is immediately outside of these edges. If *absolute* is non-`nil`, *pixelwise* is implicitly non-`nil` too.

`window-body-edges` **&optional** *window* [Function]
This function returns the edges of *window*'s body (see Section 29.4 [Window Sizes], page 701). Calling `(window-body-edges window)` is equivalent to calling `(window-edges window t)`, see above.

The following functions can be used to relate a set of frame-relative coordinates to a window:

`window-at` *x y* **&optional** *frame* [Function]
This function returns the live window at the coordinates *x* and *y* given in default character sizes (see Section 30.3.2 [Frame Font], page 799) relative to the native position of *frame* (see Section 30.3 [Frame Geometry], page 793).

If there is no window at that position, the return value is `nil`. If *frame* is omitted or `nil`, it defaults to the selected frame.

`coordinates-in-window-p` *coordinates window* [Function]
This function checks whether a window *window* occupies the frame relative coordinates *coordinates*, and if so, which part of the window that is. *window* should be a live window.

coordinates should be a cons cell of the form `(x . y)`, where *x* and *y* are given in default character sizes (see Section 30.3.2 [Frame Font], page 799) relative to the native position of *window*'s frame (see Section 30.3 [Frame Geometry], page 793).

If there is no window at the specified position, the return value is `nil` . Otherwise, the return value is one of the following:

`(relx . rely)`
: The coordinates are inside *window*. The numbers *relx* and *rely* are the equivalent window-relative coordinates for the specified position, counting from 0 at the top left corner of the window.

`mode-line`
: The coordinates are in the mode line of *window*.

`header-line`
: The coordinates are in the header line of *window*.

`tab-line` The coordinates are in the tab line of *window*.

`right-divider`
: The coordinates are in the divider separating *window* from a window on the right.

`bottom-divider`
: The coordinates are in the divider separating *window* from a window beneath.

`vertical-line`
: The coordinates are in the vertical line between *window* and its neighbor to the right. This value occurs only if the window doesn't have a scroll bar; positions in a scroll bar are considered outside the window for these purposes.

`left-fringe`
`right-fringe`
: The coordinates are in the left or right fringe of the window.

`left-margin`
`right-margin`
: The coordinates are in the left or right margin of the window.

`nil` The coordinates are not in any part of *window*.

The function `coordinates-in-window-p` does not require a frame as argument because it always uses the frame that *window* is on.

The following functions return window positions in pixels, rather than character units. Though mostly useful on graphical displays, they can also be called on text terminals, where the screen area of each text character is taken to be one pixel.

`window-pixel-edges` **&optional** *window* [Function]
: This function returns a list of pixel coordinates for the edges of *window*. Calling `(window-pixel-edges window)` is equivalent to calling `(window-edges window nil nil t)`, see above.

`window-body-pixel-edges` **&optional** *window* [Function]
: This function returns the pixel edges of *window*'s body. Calling `(window-body-pixel-edges window)` is equivalent to calling `(window-edges window t nil t)`, see above.

The following functions return window positions in pixels, relative to the origin of the display screen rather than that of the frame:

`window-absolute-pixel-edges` **&optional** *window* [Function]
: This function returns the pixel coordinates of *window* relative to an origin at (0, 0) of the display of *window*'s frame. Calling `(window-absolute-pixel-edges)` is equivalent to calling `(window-edges window nil t t)`, see above.

`window-absolute-body-pixel-edges` **&optional** *window* [Function]

This function returns the pixel coordinates of *window*'s body relative to an origin at (0, 0) of the display of *window*'s frame. Calling `(window-absolute-body-pixel-edges window)` is equivalent to calling `(window-edges window t t t)`, see above.

Combined with `set-mouse-absolute-pixel-position`, this function can be used to move the mouse pointer to an arbitrary buffer position visible in some window:

```
(let ((edges (window-absolute-body-pixel-edges))
      (position (pos-visible-in-window-p nil nil t)))
  (set-mouse-absolute-pixel-position
   (+ (nth 0 edges) (nth 0 position))
   (+ (nth 1 edges) (nth 1 position))))
```

On a graphical terminal this form "warps" the mouse cursor to the upper left corner of the glyph at the selected window's point. A position calculated this way can be also used to show a tooltip window there.

The following function returns the screen coordinates of a buffer position visible in a window:

`window-absolute-pixel-position` **&optional** *position window* [Function]

If the buffer position *position* is visible in window *window*, this function returns the display coordinates of the upper/left corner of the glyph at *position*. The return value is a cons of the X- and Y-coordinates of that corner, relative to an origin at (0, 0) of *window*'s display. It returns `nil` if *position* is not visible in *window*.

window must be a live window and defaults to the selected window. *position* defaults to the value of `window-point` of *window*.

This means that in order to move the mouse pointer to the position of point in the selected window, it's sufficient to write:

```
(let ((position (window-absolute-pixel-position)))
  (set-mouse-absolute-pixel-position
   (car position) (cdr position)))
```

The following function returns the largest rectangle that can be inscribed in a window without covering text displayed in that window.

`window-largest-empty-rectangle` **&optional** *window count min-width min-height positions left* [Function]

This function calculates the dimensions of the largest empty rectangle that can be inscribed in the specified *window*'s text area. *window* must be a live window and defaults to the selected one.

The return value is a triple of the width and the start and end y-coordinates of the largest rectangle that can be inscribed into the empty space (space not displaying any text) of the text area of *window*. No x-coordinates are returned by this function—any such rectangle is assumed to end at the right edge of *window*'s text area. If no empty space can be found, the return value is `nil`.

The optional argument *count*, if non-`nil`, specifies a maximum number of rectangles to return. This means that the return value is a list of triples specifying rectangles

with the largest rectangle first. *count* can be also a cons cell whose car specifies the number of rectangles to return and whose CDR, if non-`nil`, states that all rectangles returned must be disjoint.

The optional arguments *min-width* and *min-height*, if non-`nil`, specify the minimum width and height of any rectangle returned.

The optional argument *positions*, if non-`nil`, is a cons cell whose CAR specifies the uppermost and whose CDR specifies the lowermost pixel position that must be covered by any rectangle returned. These positions measure from the start of the text area of *window*.

The optional argument *left*, if non-`nil`, means to return values suitable for buffers displaying right to left text. In that case, any rectangle returned is assumed to start at the left edge of *window*'s text area.

Note that this function has to retrieve the dimensions of each line of *window*'s glyph matrix via `window-lines-pixel-dimensions` (see Section 41.10 [Size of Displayed Text], page 1161). Hence, this function may also return `nil` when the current glyph matrix of *window* is not up-to-date.

29.25 Mouse Window Auto-selection

The following option allows to automatically select the window under the mouse pointer. This accomplishes a policy similar to that of window managers that give focus to a frame (and thus trigger its subsequent selection) whenever the mouse pointer enters its window-system window (see Section 30.10 [Input Focus], page 827).

`mouse-autoselect-window` [User Option]

If this variable is non-`nil`, Emacs will try to automatically select the window under the mouse pointer. The following values are meaningful:

A positive number
: This specifies a delay in seconds after which auto-selection triggers. The window under the mouse pointer is selected after the mouse has remained in it for the entire duration of the delay.

A negative number
: A negative number has a similar effect as a positive number, but selects the window under the mouse pointer only after the mouse pointer has remained in it for the entire duration of the absolute value of that number and in addition has stopped moving.

Other value
: Any other non-`nil` value means to select a window instantaneously as soon as the mouse pointer enters it.

In either case, the mouse pointer must enter the text area of a window in order to trigger its selection. Dragging the scroll bar slider or the mode line of a window conceptually should not cause its auto-selection.

Mouse auto-selection selects the minibuffer window only if it is active, and never deselects the active minibuffer window.

Mouse auto-selection can be used to emulate a focus follows mouse policy for child frames (see Section 30.14 [Child Frames], page 833) which usually are not tracked by the window manager. This requires to set the value of `focus-follows-mouse` (see Section 30.10 [Input Focus], page 827) to a non-`nil` value. If the value of `focus-follows-mouse` is `auto-raise`, entering a child frame with the mouse will raise it automatically above all other child frames of that frame's parent frame.

29.26 Window Configurations

A *window configuration* records the entire layout of one frame—all windows, their sizes, their decorations, which buffers they contain, how those buffers are scrolled, and their value of point, It also includes the value of `minibuffer-scroll-window`. As a special exception, the window configuration does not record the value of point in the selected window for the current buffer.

You can bring back an entire frame layout by restoring a previously saved window configuration. If you want to record the layout of all frames instead of just one, use a frame configuration instead of a window configuration. See Section 30.13 [Frame Configurations], page 833.

`current-window-configuration` **&optional** *frame* [Function]

This function returns a new object representing *frame*'s current window configuration. The default for *frame* is the selected frame. The variable `window-persistent-parameters` specifies which window parameters (if any) are saved by this function. See Section 29.27 [Window Parameters], page 779.

`set-window-configuration` *configuration* **&optional** *dont-set-frame* *dont-set-miniwindow* [Function]

This function restores the configuration of windows and buffers as specified by *configuration*, for the frame that *configuration* was created for, regardless of whether that frame is selected or not. The argument *configuration* must be a value that was previously returned by `current-window-configuration` for that frame. Normally the function also selects the frame which is recorded in the configuration, but if *dont-set-frame* is non-`nil`, it leaves selected the frame which was already selected at the start of the function.

Normally the function restores the saved minibuffer (if any), but if *dont-set-miniwindow* is non-`nil`, the minibuffer current at the start of the function (if any) remains in the mini-window.

If the frame from which *configuration* was saved is dead, all this function does is to restore the value of the variable `minibuffer-scroll-window` and to adjust the value returned by `minibuffer-selected-window`. In this case, the function returns `nil`. Otherwise, it returns `t`.

If the buffer of a window of *configuration* has been killed since *configuration* was made, that window is, as a rule, removed from the restored configuration. However, if that window is the last window remaining in the restored configuration, another live buffer is shown in it.

Here is a way of using this function to get the same effect as `save-window-excursion`:

```
(let ((config (current-window-configuration)))
  (unwind-protect
      (progn (split-window-below nil)
             ...)
    (set-window-configuration config)))
```

`save-window-excursion` *forms...* [Macro]
This macro records the window configuration of the selected frame, executes *forms* in sequence, then restores the earlier window configuration. The return value is the value of the final form in *forms*.

Most Lisp code should not use this macro; `save-selected-window` is typically sufficient. In particular, this macro cannot reliably prevent the code in *forms* from opening new windows, because new windows might be opened in other frames (see Section 29.13.1 [Choosing Window], page 726), and `save-window-excursion` only saves and restores the window configuration on the current frame.

`window-configuration-p` *object* [Function]
This function returns `t` if *object* is a window configuration.

`window-configuration-equal-p` *config1 config2* [Function]
This function says whether two window configurations have the same window layout, but ignores the values of point and the saved scrolling positions—it can return `t` even if those aspects differ.

`window-configuration-frame` *config* [Function]
This function returns the frame for which the window configuration *config* was made.

Other primitives to look inside of window configurations would make sense, but are not implemented because we did not need them. See the file `winner.el` for some more operations on windows configurations.

The objects returned by `current-window-configuration` die together with the Emacs process. In order to store a window configuration on disk and read it back in another Emacs session, you can use the functions described next. These functions are also useful to clone the state of a frame into an arbitrary live window (`set-window-configuration` effectively clones the windows of a frame into the root window of that very frame only).

`window-state-get` **&optional** *window writable* [Function]
This function returns the state of *window* as a Lisp object. The argument *window* must be a valid window and defaults to the root window of the selected frame.

If the optional argument *writable* is non-`nil`, this means to not use markers for sampling positions like `window-point` or `window-start`. This argument should be non-`nil` when the state will be written to disk and read back in another session.

Together, the argument *writable* and the variable `window-persistent-parameters` specify which window parameters are saved by this function. See Section 29.27 [Window Parameters], page 779.

The value returned by `window-state-get` can be used in the same session to make a clone of a window in another window. It can be also written to disk and read back in another session. In either case, use the following function to restore the state of the window.

`window-state-put` *state* **&optional** *window ignore* [Function]
: This function puts the window state *state* into *window*. The argument *state* should be the state of a window returned by an earlier invocation of `window-state-get`, see above. The optional argument *window* can be either a live window or an internal window (see Section 29.2 [Windows and Frames], page 694). If *window* is not a live window, it is replaced by a new live window created on the same frame before putting *state* into it. If *window* is `nil`, it puts the window state into a new window.

  If the optional argument *ignore* is non-`nil`, it means to ignore minimum window sizes and fixed-size restrictions. If *ignore* is `safe`, this means windows can get as small as one line and/or two columns.

The functions `window-state-get` and `window-state-put` also allow to exchange the contents of two live windows. The following function does precisely that:

`window-swap-states` **&optional** *window-1 window-2 size* [Command]
: This command swaps the states of the two live windows *window-1* and *window-2*. *window-1* must specify a live window and defaults to the selected one. *window-2* must specify a live window and defaults to the window following *window-1* in the cyclic ordering of windows, excluding minibuffer windows and including live windows on all visible frames.

  Optional argument *size* non-`nil` means to try swapping the sizes of *window-1* and *window-2* as well. A value of `height` means to swap heights only, a value of `width` means to swap widths only, while `t` means to swap both widths and heights, if possible. Frames are not resized by this function.

29.27 Window Parameters

This section describes the window parameters that can be used to associate additional information with windows.

`window-parameter` *window parameter* [Function]
: This function returns *window*'s value for *parameter*. The default for *window* is the selected window. If *window* has no setting for *parameter*, this function returns `nil`.

`window-parameters` **&optional** *window* [Function]
: This function returns all parameters of *window* and their values. The default for *window* is the selected window. The return value is either `nil`, or an association list whose elements have the form `(`*`parameter`*` . `*`value`*`)`.

`set-window-parameter` *window parameter value* [Function]
: This function sets *window*'s value of *parameter* to *value* and returns *value*. The default for *window* is the selected window.

By default, the functions that save and restore window configurations or the states of windows (see Section 29.26 [Window Configurations], page 777) do not care about window parameters. This means that when you change the value of a parameter within the body of a `save-window-excursion`, the previous value is not restored when that macro exits. It also means that when you restore via `window-state-put` a window state saved earlier by `window-state-get`, all cloned windows have their parameters reset to `nil`. The following variable allows you to override the standard behavior:

`window-persistent-parameters` [Variable]

This variable is an alist specifying which parameters get saved by `current-window-configuration` and `window-state-get`, and subsequently restored by `set-window-configuration` and `window-state-put`. See Section 29.26 [Window Configurations], page 777.

The CAR of each entry of this alist is a symbol specifying the parameter. The CDR should be one of the following:

`nil` This value means the parameter is saved neither by `window-state-get` nor by `current-window-configuration`.

`t` This value specifies that the parameter is saved by `current-window-configuration` and (provided its *writable* argument is `nil`) by `window-state-get`.

`writable` This means that the parameter is saved unconditionally by both `current-window-configuration` and `window-state-get`. This value should not be used for parameters whose values do not have a read syntax. Otherwise, invoking `window-state-put` in another session may fail with an `invalid-read-syntax` error.

Some functions (notably `delete-window`, `delete-other-windows` and `split-window`), may behave specially when the window specified by their *window* argument has a parameter whose name is equal to the function's name. You can override such special behavior by binding the following variable to a non-`nil` value:

`ignore-window-parameters` [Variable]

If this variable is non-`nil`, some standard functions do not process window parameters. The functions currently affected by this are `split-window`, `delete-window`, `delete-other-windows`, and `other-window`.

An application can bind this variable to a non-`nil` value around calls to these functions. If it does so, the application is fully responsible for correctly assigning the parameters of all involved windows when exiting that function.

The following parameters are currently used by the window management code:

`delete-window`

This parameter affects the execution of `delete-window` (see Section 29.8 [Deleting Windows], page 713).

`delete-other-windows`

This parameter affects the execution of `delete-other-windows` (see Section 29.8 [Deleting Windows], page 713).

`no-delete-other-windows`

This parameter marks the window as not deletable by `delete-other-windows` (see Section 29.8 [Deleting Windows], page 713).

`split-window`

This parameter affects the execution of `split-window` (see Section 29.7 [Splitting Windows], page 710).

`other-window`
: This parameter affects the execution of `other-window` (see Section 29.10 [Cyclic Window Ordering], page 719).

`no-other-window`
: This parameter marks the window as not selectable by `other-window` (see Section 29.10 [Cyclic Window Ordering], page 719).

`clone-of`
: This parameter specifies the window that this one has been cloned from. It is installed by `window-state-get` (see Section 29.26 [Window Configurations], page 777).

`window-preserved-size`
: This parameter specifies a buffer, a direction where `nil` means vertical and `t` horizontal, and a size in pixels. If this window displays the specified buffer and its size in the indicated direction equals the size specified by this parameter, then Emacs will try to preserve the size of this window in the indicated direction. This parameter is installed and updated by the function `window-preserve-size` (see Section 29.6 [Preserving Window Sizes], page 709).

`quit-restore`
: This parameter is installed by the buffer display functions (see Section 29.13.1 [Choosing Window], page 726) and consulted by `quit-restore-window` (see Section 29.16 [Quitting Windows], page 752). It is a list of four elements, see the description of `quit-restore-window` in Section 29.16 [Quitting Windows], page 752, for details.

`window-side`
`window-slot`
: These parameters are used internally for implementing side windows (see Section 29.17 [Side Windows], page 754).

`window-atom`
: This parameter is used internally for implementing atomic windows, see Section 29.18 [Atomic Windows], page 759.

`mode-line-format`
: This parameter replaces the value of the buffer-local variable `mode-line-format` (see Section 24.4.1 [Mode Line Basics], page 551) of this window's buffer whenever this window is displayed. The symbol `none` means to suppress display of a mode line for this window. Display and contents of the mode line on other windows showing this buffer are not affected.

`header-line-format`
: This parameter replaces the value of the buffer-local variable `header-line-format` (see Section 24.4.1 [Mode Line Basics], page 551) of this window's buffer whenever this window is displayed. The symbol `none` means to suppress display of a header line for this window. Display and contents of the header line on other windows showing this buffer are not affected.

`tab-line-format`
: This parameter replaces the value of the buffer-local variable `tab-line-format` (see Section 24.4.1 [Mode Line Basics], page 551) of this window's buffer when-

ever this window is displayed. The symbol `none` means to suppress display of a tab line for this window. Display and contents of the tab line on other windows showing this buffer are not affected.

`min-margins`
: The value of this parameter is a cons cell whose CAR and CDR, if non-`nil`, specify the minimum values (in columns) for the left and right margin of this window (see Section 41.16.5 [Display Margins], page 1209. When present, Emacs will use these values instead of the actual margin widths for determining whether a window can be split or shrunk horizontally.

  Emacs never auto-adjusts the margins of any window after splitting or resizing it. It is the sole responsibility of any application setting this parameter to adjust the margins of this window as well as those of any new window that inherits this window's margins due to a split. Both `window-configuration-change-hook` and `window-size-change-functions` (see Section 29.28 [Window Hooks], page 782) should be employed for this purpose.

  This parameter was introduced in Emacs version 25.1 to support applications that use large margins to center buffer text within a window and should be used, with due care, exclusively by those applications. It might be replaced by an improved solution in future versions of Emacs.

29.28 Hooks for Window Scrolling and Changes

This section describes how Lisp programs can take action after a window has been scrolled or other window modifications occurred. We first consider the case where a window shows a different part of its buffer.

`window-scroll-functions` [Variable]
: This variable holds a list of functions that Emacs should call before redisplaying a window with scrolling. Displaying a different buffer in a window and making a new window also call these functions.

  This variable is not a normal hook, because each function is called with two arguments: the window, and its new display-start position. At the time of the call, the display-start position of the argument window is already set to its new value, and the buffer to be displayed in the window is set as the current buffer.

  These functions must take care when using `window-end` (see Section 29.20 [Window Start and End], page 762); if you need an up-to-date value, you must use the *update* argument to ensure you get it.

  Warning: don't use this feature to alter the way the window is scrolled. It's not designed for that, and such use probably won't work.

In addition, you can use `jit-lock-register` to register a Font Lock fontification function, which will be called whenever parts of a buffer are (re)fontified because a window was scrolled or its size changed. See Section 24.6.4 [Other Font Lock Variables], page 572.

The remainder of this section covers six hooks that are called during redisplay provided a significant, non-scrolling change of a window has been detected. For simplicity, these hooks and the functions they call will be collectively referred to as *window change functions*. As

any hook, these hooks can be set either globally or buffer-locally via the *local* argument of `add-hook` (see Section 24.1.2 [Setting Hooks], page 526) when the hook is installed.

The first of these hooks is run after a *window buffer change* is detected, which means that a window was created, deleted or assigned another buffer.

`window-buffer-change-functions` [Variable]

This variable specifies functions called during redisplay when window buffers have changed. The value should be a list of functions that take one argument.

Functions specified buffer-locally are called for any window showing the corresponding buffer if that window has been created or assigned that buffer since the last time window change functions were run. In this case the window is passed as argument.

Functions specified by the default value are called for a frame if at least one window on that frame has been added, deleted or assigned another buffer since the last time window change functions were run. In this case the frame is passed as argument.

The second of these hooks is run when a *window size change* has been detected which means that a window was created, assigned another buffer, or changed its total size or that of its text area.

`window-size-change-functions` [Variable]

This variable specifies functions called during redisplay when a window size change occurred. The value should be a list of functions that take one argument.

Functions specified buffer-locally are called for any window showing the corresponding buffer if that window has been added or assigned another buffer or changed its total or body size since the last time window change functions were run. In this case the window is passed as argument.

Functions specified by the default value are called for a frame if at least one window on that frame has been added or assigned another buffer or changed its total or body size since the last time window change functions were run. In this case the frame is passed as argument.

The third of these hooks is run when a *window selection change* has selected another window since the last redisplay.

`window-selection-change-functions` [Variable]

This variable specifies functions called during redisplay when the selected window or a frame's selected window has changed. The value should be a list of functions that take one argument.

Functions specified buffer-locally are called for any window showing the corresponding buffer if that window has been selected or deselected (among all windows or among all windows on its frame) since the last time window change functions were run. In this case the window is passed as argument.

Functions specified by the default value are called for a frame if that frame has been selected or deselected or the frame's selected window has changed since the last time window change functions were run. In this case the frame is passed as argument.

The fourth of these hooks is run when a *window state change* has been detected, which means that at least one of the three preceding window changes has occurred.

`window-state-change-functions` [Variable]

This variable specifies functions called during redisplay when a window buffer or size change occurred or the selected window or a frame's selected window has changed. The value should be a list of functions that take one argument.

Functions specified buffer-locally are called for any window showing the corresponding buffer if that window has been added or assigned another buffer, changed its total or body size or has been selected or deselected (among all windows or among all windows on its frame) since the last time window change functions were run. In this case the window is passed as argument.

Functions specified by the default value are called for a frame if at least one window on that frame has been added, deleted or assigned another buffer, changed its total or body size or that frame has been selected or deselected or the frame's selected window has changed since the last time window change functions were run. In this case the frame is passed as argument.

Functions specified by the default value are also run for a frame when that frame's window state change flag (see below) has been set since last redisplay.

The fifth of these hooks is run when a *window configuration change* has been detected which means that either the buffer or the size of a window changed. It differs from the four preceding hooks in the way it is run.

`window-configuration-change-hook` [Variable]

This variable specifies functions called during redisplay when either the buffer or the size of a window has changed. The value should be a list of functions that take no argument.

Functions specified buffer-locally are called for any window showing the corresponding buffer if at least one window on that frame has been added, deleted or assigned another buffer or changed its total or body size since the last time window change functions were run. Each call is performed with the window showing the buffer temporarily selected and its buffer current.

Functions specified by the default value are called for each frame if at least one window on that frame has been added, deleted or assigned another buffer or changed its total or body size since the last time window change functions were run. Each call is performed with the frame temporarily selected and the selected window's buffer current.

Finally, Emacs runs a normal hook that generalizes the behavior of `window-state-change-functions`.

`window-state-change-hook` [Variable]

The default value of this variable specifies functions called during redisplay when a window state change has been detected or the window state change flag has been set on at least one frame. The value should be a list of functions that take no argument.

Applications should put a function on this hook only if they want to react to changes that happened on (or have been signaled for) two or more frames since last redisplay. In every other case, putting the function on `window-state-change-functions` should be preferred.

Window change functions are called during redisplay for each frame as follows: First, any buffer-local window buffer change function, window size change function, selected window change and window state change functions are called in this order. Next, the default values for these functions are called in the same order. Then any buffer-local window configuration change functions are called followed by functions specified by the default value of those functions. Finally, functions on `window-state-change-hook` are run.

Window change functions are run for a specific frame only if a corresponding change was registered for that frame earlier. Such changes include the creation or deletion of a window or the assignment of another buffer or size to a window. Note that even when such a change has been registered, this does not mean that any of the hooks described above is run. If, for example, a change was registered within the scope of a window excursion (see Section 29.26 [Window Configurations], page 777), this will trigger a call of window change functions only if that excursion still persists at the time change functions are run. If it is exited earlier, hooks will be run only if registered by a change outside the scope of that excursion.

The *window state change flag* of a frame, if set, will cause the default values of `window-state-change-functions` (for that frame) and `window-state-change-hook` to be run during next redisplay regardless of whether a window state change actually occurred for that frame or not. After running any functions on these hooks, the flag is reset for each frame. Applications can set that flag and inspect its value using the following functions.

`set-frame-window-state-change` **&optional** *frame arg* [Function]
: This function sets *frame*'s window state change flag if *arg* is non-`nil` and resets it otherwise. *frame* must be a live frame and defaults to the selected one.

`frame-window-state-change` **&optional** *frame* [Function]
: This functions returns `t` if *frame*'s window state change flag is set and `nil` otherwise. *frame* must be a live frame and defaults to the selected one.

While window change functions are run, the functions described next can be called to get more insight into what has changed for a specific window or frame since the last redisplay. All these functions take a live window as single, optional argument, defaulting to the selected window.

`window-old-buffer` **&optional** *window* [Function]
: This function returns the buffer shown in *window* at the last time window change functions were run for *window*'s frame. If it returns `nil`, *window* has been created after that. If it returns `t`, *window* was not shown at that time but has been restored from a previously saved window configuration afterwards. Otherwise, the return value is the buffer shown by *window* at that time.

`window-old-pixel-width` **&optional** *window* [Function]
: This function returns the total pixel width of *window* the last time window change functions found *window* live on its frame. It is zero if *window* was created after that.

`window-old-pixel-height` **&optional** *window* [Function]
: This function returns the total pixel height of *window* the last time window change functions found *window* live on its frame. It is zero if *window* was created after that.

`window-old-body-pixel-width` **&optional** *window* [Function]
: This function returns the pixel width of *window*'s text area the last time window change functions found *window* live on its frame. It is zero if *window* was created after that.

`window-old-body-pixel-height` **&optional** *window* [Function]
: This function returns the pixel height of *window*'s text area the last time window change functions found *window* live on its frame. It is zero if *window* was created after that.

In order to find out which window or frame was selected the last time window change functions were run, the following functions can be used:

`frame-old-selected-window` **&optional** *frame* [Function]
: This function returns the selected window of *frame* at the last time window change functions were run. If omitted or `nil` *frame* defaults to the selected frame.

`old-selected-window` [Function]
: This function returns the selected window at the last time window change functions were run.

`old-selected-frame` [Function]
: This function returns the selected frame at the last time window change functions were run.

Note that window change functions provide no information about which windows have been deleted since the last time they were run. If necessary, applications should remember any window showing a specific buffer in a local variable of that buffer and update it in a function run by the default values of any of the hooks that are run when a window buffer change was detected.

The following caveats should be considered when adding a function to window change functions:

- Some operations will not trigger a call of window change functions. These include showing another buffer in a minibuffer window or any change of a tooltip window.
- Window change functions should not create or delete windows or change the buffer, size or selection status of any window because there is no guarantee that the information about such a change will be propagated to other window change functions. If at all, any such change should be executed only by the last function listed by the default value of `window-state-change-hook`.
- Macros like `save-window-excursion`, `with-selected-window` or `with-current-buffer` can be used when running window change functions.
- Running window change functions does not save and restore match data. Unless running `window-configuration-change-hook` it does not save or restore the selected window or frame or the current buffer either.
- Any redisplay triggering the run of window change functions may be aborted. If the abort occurs before window change functions have run to their completion, they will be run again with the previous values, that is, as if redisplay had not been performed. If aborted later, they will be run with the new values, that is, as if redisplay had been actually performed.

30 Frames

A *frame* is a screen object that contains one or more Emacs windows (see Chapter 29 [Windows], page 692). It is the kind of object called a "window" in the terminology of graphical environments; but we can't call it a "window" here, because Emacs uses that word in a different way. In Emacs Lisp, a *frame object* is a Lisp object that represents a frame on the screen. See Section 2.5.4 [Frame Type], page 27.

A frame initially contains a single main window and/or a minibuffer window; you can subdivide the main window vertically or horizontally into smaller windows. See Section 29.7 [Splitting Windows], page 710.

A *terminal* is a display device capable of displaying one or more Emacs frames. In Emacs Lisp, a *terminal object* is a Lisp object that represents a terminal. See Section 2.5.5 [Terminal Type], page 27.

There are two classes of terminals: *text terminals* and *graphical terminals*. Text terminals are non-graphics-capable displays, including `xterm` and other terminal emulators. On a text terminal, each Emacs frame occupies the terminal's entire screen; although you can create additional frames and switch between them, the terminal only shows one frame at a time. Graphical terminals, on the other hand, are managed by graphical display systems such as the X Window System, which allow Emacs to show multiple frames simultaneously on the same display.

On GNU and Unix systems, you can create additional frames on any available terminal, within a single Emacs session, regardless of whether Emacs was started on a text or graphical terminal. Emacs can display on both graphical and text terminals simultaneously. This comes in handy, for instance, when you connect to the same session from several remote locations. See Section 30.2 [Multiple Terminals], page 789.

`framep` *object* [Function]

This predicate returns a non-`nil` value if *object* is a frame, and `nil` otherwise. For a frame, the value indicates which kind of display the frame uses:

`t`	The frame is displayed on a text terminal.
`x`	The frame is displayed on an X graphical terminal.
`w32`	The frame is displayed on a MS-Windows graphical terminal.
`ns`	The frame is displayed on a GNUstep or Macintosh Cocoa graphical terminal.
`pc`	The frame is displayed on an MS-DOS terminal.
`pgtk`	The frame is displayed using pure GTK facilities.

`frame-terminal` **&optional** *frame* [Function]

This function returns the terminal object that displays *frame*. If *frame* is `nil` or unspecified, it defaults to the selected frame.

`terminal-live-p` *object* [Function]

This predicate returns a non-`nil` value if *object* is a terminal that is live (i.e., not deleted), and `nil` otherwise. For live terminals, the return value indicates what kind of frames are displayed on that terminal; the list of possible values is the same as for `framep` above.

On a graphical terminal we distinguish two types of frames: A normal *top-level frame* is a frame whose window-system window is a child of the window-system's root window for that terminal. A child frame is a frame whose window-system window is the child of the window-system window of another Emacs frame. See Section 30.14 [Child Frames], page 833.

30.1 Creating Frames

To create a new frame, call the function `make-frame`.

`make-frame` **&optional** *parameters* [Command]

This function creates and returns a new frame, displaying the current buffer.

The *parameters* argument is an alist that specifies frame parameters for the new frame. See Section 30.4 [Frame Parameters], page 804. If you specify the `terminal` parameter in *parameters*, the new frame is created on that terminal. Otherwise, if you specify the `window-system` frame parameter in *parameters*, that determines whether the frame should be displayed on a text terminal or a graphical terminal. See Section 41.25 [Window Systems], page 1253. If neither is specified, the new frame is created in the same terminal as the selected frame.

Any parameters not mentioned in *parameters* default to the values in the alist `default-frame-alist` (see Section 30.4.2 [Initial Parameters], page 806); parameters not specified there default from the X resources or its equivalent on your operating system (see Section "X Resources" in *The GNU Emacs Manual*). After the frame is created, this function applies any parameters specified in `frame-inherited-parameters` (see below) it has no assigned yet, taking the values from the frame that was selected when `make-frame` was called.

Note that on multi-monitor displays (see Section 30.2 [Multiple Terminals], page 789), the window manager might position the frame differently than specified by the positional parameters in *parameters* (see Section 30.4.3.2 [Position Parameters], page 807). For example, some window managers have a policy of displaying the frame on the monitor that contains the largest part of the window (a.k.a. the *dominating* monitor).

This function itself does not make the new frame the selected frame. See Section 30.10 [Input Focus], page 827. The previously selected frame remains selected. On graphical terminals, however, the window system may select the new frame for its own reasons.

`before-make-frame-hook` [Variable]

A normal hook run by `make-frame` before it creates the frame.

`after-make-frame-functions` [Variable]

An abnormal hook run by `make-frame` after it created the frame. Each function in `after-make-frame-functions` receives one argument, the frame just created.

Note that any functions added to these hooks by your initial file are usually not run for the initial frame, since Emacs reads the initial file only after creating that frame. However, if the initial frame is specified to use a separate minibuffer frame (see Section 30.9 [Minibuffers and Frames], page 826), the functions will be run for both, the minibuffer-less and the minibuffer frame. Alternatively, you can add functions to these hooks in your "early init

file" (see Section 42.1.2 [Init File], page 1263), in which case they will be in effect for the initial frame as well.

`frame-inherited-parameters` [Variable]
: This variable specifies the list of frame parameters that a newly created frame inherits from the currently selected frame. For each parameter (a symbol) that is an element in this list and has not been assigned earlier when processing `make-frame`, the function sets the value of that parameter in the created frame to its value in the selected frame.

`server-after-make-frame-hook` [User Option]
: A normal hook run when the Emacs server starts using a client frame. When this hook is called, the client frame is the selected one. Note that, depending on how `emacsclient` was invoked (see Section "Invoking emacsclient" in *The GNU Emacs Manual*), this client frame could be a new frame created for the client, or it could be an existing frame that the server reused for handling the client commands. See Section "Emacs Server" in *The GNU Emacs Manual*.

30.2 Multiple Terminals

Emacs represents each terminal as a *terminal object* data type (see Section 2.5.5 [Terminal Type], page 27). On GNU and Unix systems, Emacs can use multiple terminals simultaneously in each session. On other systems, it can only use a single terminal. Each terminal object has the following attributes:

- The name of the device used by the terminal (e.g., '`:0.0`' or `/dev/tty`).
- The terminal and keyboard coding systems used on the terminal. See Section 34.10.8 [Terminal I/O Encoding], page 993.
- The kind of display associated with the terminal. This is the symbol returned by the function `terminal-live-p` (i.e., `x`, `t`, `w32`, `ns`, `pc`, `haiku`, or `pgtk`). See Chapter 30 [Frames], page 787.
- A list of terminal parameters. See Section 30.5 [Terminal Parameters], page 823.

There is no primitive for creating terminal objects. Emacs creates them as needed, such as when you call `make-frame-on-display` (described below).

`terminal-name` **&optional** *terminal* [Function]
: This function returns the file name of the device used by *terminal*. If *terminal* is omitted or `nil`, it defaults to the selected frame's terminal. *terminal* can also be a frame, meaning that frame's terminal.

`terminal-list` [Function]
: This function returns a list of all live terminal objects.

`get-device-terminal` *device* [Function]
: This function returns a terminal whose device name is given by *device*. If *device* is a string, it can be either the file name of a terminal device, or the name of an X display of the form '*`host:server.screen`*'. If *device* is a frame, this function returns that frame's terminal; `nil` means the selected frame. Finally, if *device* is a terminal object that represents a live terminal, that terminal is returned. The function signals an error if its argument is none of the above.

`delete-terminal` **&optional** *terminal force* [Function]
This function deletes all frames on *terminal* and frees the resources used by it. It runs the abnormal hook `delete-terminal-functions`, passing *terminal* as the argument to each function.

If *terminal* is omitted or `nil`, it defaults to the selected frame's terminal. *terminal* can also be a frame, meaning that frame's terminal.

Normally, this function signals an error if you attempt to delete the sole active terminal, but if *force* is non-`nil`, you are allowed to do so. Emacs automatically calls this function when the last frame on a terminal is deleted (see Section 30.7 [Deleting Frames], page 824).

`delete-terminal-functions` [Variable]
An abnormal hook run by `delete-terminal`. Each function receives one argument, the *terminal* argument passed to `delete-terminal`. Due to technical details, the functions may be called either just before the terminal is deleted, or just afterwards.

A few Lisp variables are *terminal-local*; that is, they have a separate binding for each terminal. The binding in effect at any time is the one for the terminal that the currently selected frame belongs to. These variables include `default-minibuffer-frame`, `defining-kbd-macro`, `last-kbd-macro`, and `system-key-alist`. They are always terminal-local, and can never be buffer-local (see Section 12.11 [Buffer-Local Variables], page 207).

On GNU and Unix systems, each X display is a separate graphical terminal. When Emacs is started from within the X window system, it uses the X display specified by the `DISPLAY` environment variable, or by the '`--display`' option (see Section "Initial Options" in *The GNU Emacs Manual*). Emacs can connect to other X displays via the command `make-frame-on-display`. Each X display has its own selected frame and its own minibuffer windows; however, only one of those frames is *the* selected frame at any given moment (see Section 30.10 [Input Focus], page 827). Emacs can even connect to other text terminals, by interacting with the `emacsclient` program. See Section "Emacs Server" in *The GNU Emacs Manual*.

A single X server can handle more than one display. Each X display has a three-part name, '*`hostname:displaynumber.screennumber`*'. The first part, *hostname*, specifies the name of the machine to which the display is physically connected. The second part, *displaynumber*, is a zero-based number that identifies one or more monitors connected to that machine that share a common keyboard and pointing device (mouse, tablet, etc.). The third part, *screennumber*, identifies a zero-based screen number (a separate monitor) that is part of a single monitor collection on that X server. When you use two or more screens belonging to one server, Emacs knows by the similarity in their names that they share a single keyboard.

Systems that don't use the X window system, such as MS-Windows, don't support the notion of X displays, and have only one display on each host. The display name on these systems doesn't follow the above 3-part format; for example, the display name on MS-Windows systems is a constant string '`w32`', and exists for compatibility, so that you could pass it to functions that expect a display name.

`make-frame-on-display` *display* **&optional** *parameters* [Command]
This function creates and returns a new frame on *display*, taking the other frame parameters from the alist *parameters*. *display* should be the name of an X display (a string).

Before creating the frame, this function ensures that Emacs is set up to display graphics. For instance, if Emacs has not processed X resources (e.g., if it was started on a text terminal), it does so at this time. In all other respects, this function behaves like `make-frame` (see Section 30.1 [Creating Frames], page 788).

`x-display-list` [Function]
This function returns a list that indicates which X displays Emacs has a connection to. The elements of the list are strings, and each one is a display name.

`x-open-connection` *display* **&optional** *xrm-string must-succeed* [Function]
This function opens a connection to the X display *display*, without creating a frame on that display. Normally, Emacs Lisp programs need not call this function, as `make-frame-on-display` calls it automatically. The only reason for calling it is to check whether communication can be established with a given X display.

The optional argument *xrm-string*, if not `nil`, is a string of resource names and values, in the same format used in the `.Xresources` file. See Section "X Resources" in *The GNU Emacs Manual*. These values apply to all Emacs frames created on this display, overriding the resource values recorded in the X server. Here's an example of what this string might look like:

```
"*BorderWidth: 3\n*InternalBorder: 2\n"
```

If *must-succeed* is non-`nil`, failure to open the connection terminates Emacs. Otherwise, it is an ordinary Lisp error.

`x-close-connection` *display* [Function]
This function closes the connection to display *display*. Before you can do this, you must first delete all the frames that were open on that display (see Section 30.7 [Deleting Frames], page 824).

On some multi-monitor setups, a single X display outputs to more than one physical monitor. You can use the functions `display-monitor-attributes-list` and `frame-monitor-attributes` to obtain information about such setups.

`display-monitor-attributes-list` **&optional** *display* [Function]
This function returns a list of physical monitor attributes on *display*, which can be a display name (a string), a terminal, or a frame; if omitted or `nil`, it defaults to the selected frame's display. Each element of the list is an association list, representing the attributes of a physical monitor. The first element corresponds to the primary monitor. The attribute keys and values are:

'`geometry`'
: Position of the top-left corner of the monitor's screen and its size, in pixels, as '`(x y width height)`'. Note that, if the monitor is not the primary monitor, some of the coordinates might be negative.

'`workarea`'
: Position of the top-left corner and size of the work area (usable space) in pixels as '`(x y width height)`'. This may be different from '`geometry`' in that space occupied by various window manager features (docks, taskbars, etc.) may be excluded from the work area. Whether or not such features actually subtract from the work area depends on the platform and environment. Again, if the monitor is not the primary monitor, some of the coordinates might be negative.

'`mm-size`'
: Width and height in millimeters as '`(width height)`'

'`frames`'
: List of frames that this physical monitor dominates (see below).

'`name`'
: Name of the physical monitor as *string*.

'`source`'
: Source of the multi-monitor information as *string*; e.g., '`XRandR 1.5`', '`XRandr`' or '`Xinerama`'.

x, *y*, *width*, and *height* are integers. '`name`' and '`source`' may be absent.

A frame is *dominated* by a physical monitor when either the largest area of the frame resides in that monitor, or (if the frame does not intersect any physical monitors) that monitor is the closest to the frame. Every (non-tooltip) frame (whether visible or not) in a graphical display is dominated by exactly one physical monitor at a time, though the frame can span multiple (or no) physical monitors.

Here's an example of the data produced by this function on a 2-monitor display:

```
(display-monitor-attributes-list)
⇒
(((geometry 0 0 1920 1080) ;; Left-hand, primary monitor
  (workarea 0 0 1920 1050) ;; A taskbar occupies some of the height
  (mm-size 677 381)
  (name . "DISPLAY1")
  (frames #<frame emacs@host *Messages* 0x11578c0>
          #<frame emacs@host *scratch* 0x114b838>))
 ((geometry 1920 0 1680 1050) ;; Right-hand monitor
  (workarea 1920 0 1680 1050) ;; Whole screen can be used
  (mm-size 593 370)
  (name . "DISPLAY2")
  (frames)))
```

`frame-monitor-attributes` **&optional** *frame* [Function]
: This function returns the attributes of the physical monitor dominating (see above) *frame*, which defaults to the selected frame.

On multi-monitor displays it is possible to use the command `make-frame-on-monitor` to make frames on the specified monitor.

`make-frame-on-monitor` *monitor* **&optional** *display parameters* [Command]
: This function creates and returns a new frame on *monitor* located on *display*, taking the other frame parameters from the alist *parameters*. *monitor* should be the name of

the physical monitor, the same string as returned by the function `display-monitor-attributes-list` in the attribute `name`. *display* should be the name of an X display (a string).

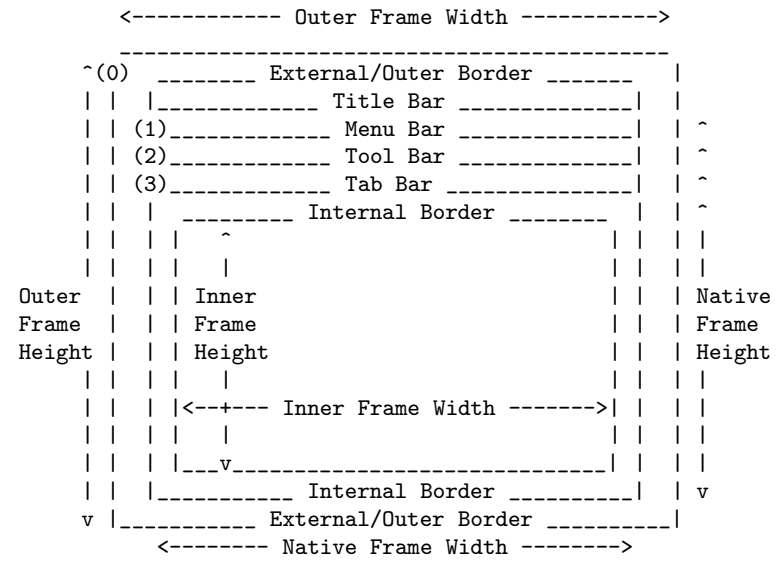
`display-monitors-changed-functions` [Variable]
This variable is an abnormal hook run when the monitor configuration changes, which can happen if a monitor is rotated, moved, added or removed from a multiple-monitor setup, if the primary monitor changes, or if the resolution of a monitor changes. It is called with a single argument consisting of the terminal on which the monitor configuration changed. Programs should call `display-monitor-attributes-list` with the terminal as the argument to retrieve the new monitor configuration on that terminal.

30.3 Frame Geometry

The geometry of a frame depends on the toolkit that was used to build this instance of Emacs and the terminal that displays the frame. This chapter describes these dependencies and some of the functions to deal with them. Note that the *frame* argument of all of these functions has to specify a live frame (see Section 30.7 [Deleting Frames], page 824). If omitted or `nil`, it specifies the selected frame (see Section 30.10 [Input Focus], page 827).

30.3.1 Frame Layout

A visible frame occupies a rectangular area on its terminal's display. This area may contain a number of nested rectangles, each serving a different purpose. The drawing below sketches the layout of a frame on a graphical terminal:



In practice not all of the areas shown in the drawing will or may be present. The meaning of these areas is described below.

Outer Frame

The *outer frame* is a rectangle comprising all areas shown in the drawing. The edges of that rectangle are called the *outer edges* of the frame. Together, the *outer width* and *outer height* of the frame specify the *outer size* of that rectangle.

Knowing the outer size of a frame is useful for fitting a frame into the working area of its display (see Section 30.2 [Multiple Terminals], page 789) or for placing two frames adjacent to each other on the screen. Usually, the outer size of a frame is available only after the frame has been mapped (made visible, see Section 30.11 [Visibility of Frames], page 831) at least once. For the initial frame or a frame that has not been created yet, the outer size can be only estimated or must be calculated from the window-system's or window manager's defaults. One workaround is to obtain the differences of the outer and native (see below) sizes of a mapped frame and use them for calculating the outer size of the new frame.

The position of the upper left corner of the outer frame (indicated by '`(0)`' in the drawing above) is the *outer position* of the frame. The outer position of a graphical frame is also referred to as "the position" of the frame because it usually remains unchanged on its display whenever the frame is resized or its layout is changed.

The outer position is specified by and can be set via the `left` and `top` frame parameters (see Section 30.4.3.2 [Position Parameters], page 807). For a normal, top-level frame these parameters usually represent its absolute position (see below) with respect to its display's origin. For a child frame (see Section 30.14 [Child Frames], page 833) these parameters represent its position relative to the native position (see below) of its parent frame. For frames on text terminals the values of these parameters are meaningless and always zero.

External Border

The *external border* is part of the decorations supplied by the window manager. It is typically used for resizing the frame with the mouse and is therefore not shown on "fullboth" and maximized frames (see Section 30.4.3.3 [Size Parameters], page 810). Its width is determined by the window manager and cannot be changed by Emacs' functions.

External borders don't exist on text terminal frames. For graphical frames, their display can be suppressed by setting the `override-redirect` or `undecorated` frame parameter (see Section 30.4.3.8 [Management Parameters], page 816).

Outer Border

The *outer border* is a separate border whose width can be specified with the `border-width` frame parameter (see Section 30.4.3.4 [Layout Parameters], page 812). In practice, either the external or the outer border of a frame are displayed but never both at the same time. Usually, the outer border is shown only for special frames that are not (fully) controlled by the window manager like tooltip frames (see Section 41.26 [Tooltips], page 1254), child frames (see Section 30.14 [Child Frames], page 833) and `undecorated` or `override-redirect` frames (see Section 30.4.3.8 [Management Parameters], page 816).

Outer borders are never shown on text terminal frames and on frames generated by GTK+ routines. On MS-Windows, the outer border is emulated with the help of a one pixel wide external border. Non-toolkit builds on X allow to change the color of the outer border by setting the `border-color` frame parameter (see Section 30.4.3.4 [Layout Parameters], page 812).

Title Bar The *title bar*, a.k.a. *caption bar*, is also part of the window manager's decorations and typically displays the title of the frame (see Section 30.6 [Frame Titles], page 823) as well as buttons for minimizing, maximizing and deleting the frame. It can be also used for dragging the frame with the mouse. The title bar is usually not displayed for fullboth (see Section 30.4.3.3 [Size Parameters], page 810), tooltip (see Section 41.26 [Tooltips], page 1254) and child frames (see Section 30.14 [Child Frames], page 833) and doesn't exist for terminal frames. Display of the title bar can be suppressed by setting the `override-redirect` or the `undecorated` frame parameters (see Section 30.4.3.8 [Management Parameters], page 816).

Menu Bar The menu bar (see Section 23.18.5 [Menu Bar], page 517) can be either internal (drawn by Emacs itself) or external (drawn by the toolkit). Most builds (GTK+, Lucid, Motif and MS-Windows) rely on an external menu bar. NS also uses an external menu bar which, however, is not part of the outer frame. Non-toolkit builds can provide an internal menu bar. On text terminal frames, the menu bar is part of the frame's root window (see Section 29.2 [Windows and Frames], page 694). As a rule, menu bars are never shown on child frames (see Section 30.14 [Child Frames], page 833). Display of the menu bar can be suppressed by setting the `menu-bar-lines` parameter (see Section 30.4.3.4 [Layout Parameters], page 812) to zero.

Whether the menu bar is wrapped or truncated whenever its width becomes too large to fit on its frame depends on the toolkit . Usually, only Motif and MS-Windows builds can wrap the menu bar. When they (un-)wrap the menu bar, they try to keep the outer height of the frame unchanged, so the native height of the frame (see below) will change instead.

Tool Bar Like the menu bar, the tool bar (see Section 23.18.6 [Tool Bar], page 518) can be either internal (drawn by Emacs itself) or external (drawn by a toolkit). The GTK+ and NS builds have the tool bar drawn by the toolkit. The remaining builds use internal tool bars. With GTK+ the tool bar can be located on either side of the frame, immediately outside the internal border, see below. Tool bars are usually not shown for child frames (see Section 30.14 [Child Frames], page 833). Display of the tool bar can be suppressed by setting the `tool-bar-lines` parameter (see Section 30.4.3.4 [Layout Parameters], page 812) to zero.

If the variable `auto-resize-tool-bars` is non-`nil`, Emacs wraps the internal tool bar when its width becomes too large for its frame. If and when Emacs (un-)wraps the internal tool bar, it by default keeps the outer height of the frame unchanged, so the native height of the frame (see below) will change instead. Emacs built with GTK+, on the other hand, never wraps the tool bar but may automatically increase the outer width of a frame in order to accommodate an overlong tool bar.

Tab Bar
: The tab bar (see Section "Tab Bars" in *The GNU Emacs Manual*) is always drawn by Emacs itself. The tab bar appears above the tool bar in Emacs built with an internal tool bar, and below the tool bar in builds with an external tool bar. Display of the tab bar can be suppressed by setting the `tab-bar-lines` parameter (see Section 30.4.3.4 [Layout Parameters], page 812) to zero.

Native Frame
: The *native frame* is a rectangle located entirely within the outer frame. It excludes the areas occupied by an external or outer border, the title bar and any external menu or tool bar. The edges of the native frame are called the *native edges* of the frame. Together, the *native width* and *native height* of a frame specify the *native size* of the frame.

  The native size of a frame is the size Emacs passes to the window-system or window manager when creating or resizing the frame from within Emacs. It is also the size Emacs receives from the window-system or window manager whenever these resize the frame's window-system window, for example, after maximizing the frame by clicking on the corresponding button in the title bar or when dragging its external border with the mouse.

  The position of the top left corner of the native frame specifies the *native position* of the frame. (1)–(3) in the drawing above indicate that position for the various builds:

  (1) non-toolkit, Haiku, and terminal frames

  (2) Lucid, Motif, and MS-Windows frames

  (3) GTK+ and NS frames

  Accordingly, the native height of a frame may include the height of the tool bar but not that of the menu bar (Lucid, Motif, MS-Windows) or those of the menu bar and the tool bar (non-toolkit and text terminal frames).

  The native position of a frame is the reference position for functions that set or return the current position of the mouse (see Section 30.16 [Mouse Position], page 838) and for functions dealing with the position of windows like `window-edges`, `window-at` or `coordinates-in-window-p` (see Section 29.24 [Coordinates and Windows], page 772). It also specifies the (0, 0) origin for locating and positioning child frames within this frame (see Section 30.14 [Child Frames], page 833).

  Note also that the native position of a frame usually remains unaltered on its display when removing or adding the window manager decorations by changing the frame's `override-redirect` or `undecorated` parameter (see Section 30.4.3.8 [Management Parameters], page 816).

Internal Border
: The internal border is a border drawn by Emacs around the inner frame (see below). The specification of its appearance depends on whether or not the given frame is a child frame (see Section 30.14 [Child Frames], page 833).

  For normal frames its width is specified by the `internal-border-width` frame parameter (see Section 30.4.3.4 [Layout Parameters], page 812), and its color is specified by the background of the `internal-border` face.

For child frames its width is specified by the `child-frame-border-width` frame parameter (but will use the `internal-border-width` parameter as fallback), and its color is specified by the background of the `child-frame-border` face.

Inner Frame
: The *inner frame* is the rectangle reserved for the frame's windows. It's enclosed by the internal border which, however, is not part of the inner frame. Its edges are called the *inner edges* of the frame. The *inner width* and *inner height* specify the *inner size* of the rectangle. The inner frame is sometimes also referred to as the *display area* of the frame.

  As a rule, the inner frame is subdivided into the frame's root window (see Section 29.2 [Windows and Frames], page 694) and the frame's minibuffer window (see Section 21.11 [Minibuffer Windows], page 419). There are two notable exceptions to this rule: A *minibuffer-less frame* contains a root window only and does not contain a minibuffer window. A *minibuffer-only frame* contains only a minibuffer window which also serves as that frame's root window. See Section 30.4.2 [Initial Parameters], page 806, for how to create such frame configurations.

Text Area
: The *text area* of a frame is a somewhat fictitious area that can be embedded in the native frame. Its position is unspecified. Its width can be obtained by removing from that of the native width the widths of the internal border, one vertical scroll bar, and one left and one right fringe if they are specified for this frame, see Section 30.4.3.4 [Layout Parameters], page 812. Its height can be obtained by removing from that of the native height the widths of the internal border and the heights of the frame's internal menu and tool bars, the tab bar and one horizontal scroll bar if specified for this frame.

The *absolute position* of a frame is given as a pair (X, Y) of horizontal and vertical pixel offsets relative to an origin (0, 0) of the frame's display. Correspondingly, the *absolute edges* of a frame are given as pixel offsets from that origin.

Note that with multiple monitors, the origin of the display does not necessarily coincide with the top-left corner of the entire usable display area of the terminal. Hence the absolute position of a frame can be negative in such an environment even when that frame is completely visible.

By convention, vertical offsets increase "downwards". This means that the height of a frame is obtained by subtracting the offset of its top edge from that of its bottom edge. Horizontal offsets increase "rightwards", as expected, so a frame's width is calculated by subtracting the offset of its left edge from that of its right edge.

For a frame on a graphical terminal the following function returns the sizes of the areas described above:

`frame-geometry` **&optional** *frame* [Function]
: This function returns geometric attributes of *frame*. The return value is an association list of the attributes listed below. All coordinate, height and width values are integers counting pixels. Note that if *frame* has not been mapped yet, (see Section 30.11 [Visibility of Frames], page 831) some of the return values may only represent ap-

proximations of the actual values—those that can be seen after the frame has been mapped.

`outer-position`
: A cons representing the absolute position of the outer *frame*, relative to the origin at position (0, 0) of *frame*'s display.

`outer-size`
: A cons of the outer width and height of *frame*.

`external-border-size`
: A cons of the horizontal and vertical width of *frame*'s external borders as supplied by the window manager. If the window manager doesn't supply these values, Emacs will try to guess them from the coordinates of the outer and inner frame.

`outer-border-width`
: The width of the outer border of *frame*. The value is meaningful for non-GTK+ X builds only.

`title-bar-size`
: A cons of the width and height of the title bar of *frame* as supplied by the window manager or operating system. If both of them are zero, the frame has no title bar. If only the width is zero, Emacs was not able to retrieve the width information.

`menu-bar-external`
: If non-`nil`, this means the menu bar is external (not part of the native frame of *frame*).

`menu-bar-size`
: A cons of the width and height of the menu bar of *frame*.

`tool-bar-external`
: If non-`nil`, this means the tool bar is external (not part of the native frame of *frame*).

`tool-bar-position`
: This tells on which side the tool bar on *frame* is and can be one of `left`, `top`, `right` or `bottom`. The only toolkit that currently supports a value other than `top` is GTK+.

`tool-bar-size`
: A cons of the width and height of the tool bar of *frame*.

`internal-border-width`
: The width of the internal border of *frame*.

The following function can be used to retrieve the edges of the outer, native and inner frame.

`frame-edges` **&optional** *frame type* [Function]
: This function returns the absolute edges of the outer, native or inner frame of *frame*. *frame* must be a live frame and defaults to the selected one. The returned list has the

form (*left top right bottom*) where all values are in pixels relative to the origin of *frame*'s display. For terminal frames the values returned for *left* and *top* are always zero.

Optional argument *type* specifies the type of the edges to return: `outer-edges` means to return the outer edges of *frame*, `native-edges` (or `nil`) means to return its native edges and `inner-edges` means to return its inner edges.

By convention, the pixels of the display at the values returned for *left* and *top* are considered to be inside (part of) *frame*. Hence, if *left* and *top* are both zero, the pixel at the display's origin is part of *frame*. The pixels at *bottom* and *right*, on the other hand, are considered to lie immediately outside *frame*. This means that if you have, for example, two side-by-side frames positioned such that the right outer edge of the frame on the left equals the left outer edge of the frame on the right, the pixels at that edge show a part of the frame on the right.

30.3.2 Frame Font

Each frame has a *default font* which specifies the default character size for that frame. This size is meant when retrieving or changing the size of a frame in terms of columns or lines (see Section 30.4.3.3 [Size Parameters], page 810). It is also used when resizing (see Section 29.4 [Window Sizes], page 701) or splitting (see Section 29.7 [Splitting Windows], page 710) windows.

The terms *line height* and *canonical character height* are sometimes used instead of "default character height". Similarly, the terms *column width* and *canonical character width* are used instead of "default character width".

`frame-char-height` **&optional** *frame* [Function]
`frame-char-width` **&optional** *frame* [Function]
These functions return the default height and width of a character in *frame*, measured in pixels. Together, these values establish the size of the default font on *frame*. The values depend on the choice of font for *frame*, see Section 30.4.3.10 [Font and Color Parameters], page 820.

The default font can be also set directly with the following function:

`set-frame-font` *font* **&optional** *keep-size frames* [Command]
This sets the default font to *font*. When called interactively, it prompts for the name of a font, and uses that font on the selected frame. When called from Lisp, *font* should be a font name (a string), a font object, font entity, or a font spec.

If the optional argument *keep-size* is `nil`, this keeps the number of frame lines and columns fixed. (If non-`nil`, the option `frame-inhibit-implied-resize` described in the next section will override this.) If *keep-size* is non-`nil` (or with a prefix argument), it tries to keep the size of the display area of the current frame fixed by adjusting the number of lines and columns.

If the optional argument *frames* is `nil`, this applies the font to the selected frame only. If *frames* is non-`nil`, it should be a list of frames to act upon, or `t` meaning all existing and all future graphical frames.

30.3.3 Frame Position

On graphical systems, the position of a normal top-level frame is specified as the absolute position of its outer frame (see Section 30.3 [Frame Geometry], page 793). The position of a child frame (see Section 30.14 [Child Frames], page 833) is specified via pixel offsets of its outer edges relative to the native position of its parent frame.

You can access or change the position of a frame using the frame parameters `left` and `top` (see Section 30.4.3.2 [Position Parameters], page 807). Here are two additional functions for working with the positions of an existing, visible frame. For both functions, the argument *frame* must denote a live frame and defaults to the selected frame.

`frame-position` **&optional** *frame* [Function]

For a normal, non-child frame this function returns a cons of the pixel coordinates of its outer position (see Section 30.3.1 [Frame Layout], page 793) with respect to the origin `(0, 0)` of its display. For a child frame (see Section 30.14 [Child Frames], page 833) this function returns the pixel coordinates of its outer position with respect to an origin `(0, 0)` at the native position of *frame*'s parent.

Negative values never indicate an offset from the right or bottom edge of *frame*'s display or parent frame. Rather, they mean that *frame*'s outer position is on the left and/or above the origin of its display or the native position of its parent frame. This usually means that *frame* is only partially visible (or completely invisible). However, on systems where the display's origin does not coincide with its top-left corner, the frame may be visible on a secondary monitor.

On a text terminal frame both values are zero.

`set-frame-position` *frame x y* [Function]

This function sets the outer frame position of *frame* to (*x*, *y*). The latter arguments specify pixels and normally count from the origin at the position (0, 0) of *frame*'s display. For child frames, they count from the native position of *frame*'s parent frame.

Negative parameter values position the right edge of the outer frame by -*x* pixels left from the right edge of the screen (or the parent frame's native rectangle) and the bottom edge by -*y* pixels up from the bottom edge of the screen (or the parent frame's native rectangle).

Note that negative values do not permit to align the right or bottom edge of *frame* exactly at the right or bottom edge of its display or parent frame. Neither do they allow to specify a position that does not lie within the edges of the display or parent frame. The frame parameters `left` and `top` (see Section 30.4.3.2 [Position Parameters], page 807) allow to do that, but may still fail to provide good results for the initial or a new frame.

This function has no effect on text terminal frames.

`move-frame-functions` [Variable]

This hook specifies the functions that are run when an Emacs frame is moved (assigned a new position) by the window-system or window manager. The functions are run with one argument, the frame that moved. For a child frame (see Section 30.14 [Child Frames], page 833), the functions are run only when the position of the frame changes in relation to that of its parent frame.

30.3.4 Frame Size

The canonical way to specify the *size of a frame* from within Emacs is by specifying its *text size*—a tuple of the width and height of the frame's text area (see Section 30.3.1 [Frame Layout], page 793). It can be measured either in pixels or in terms of the frame's canonical character size (see Section 30.3.2 [Frame Font], page 799).

For frames with an internal menu or tool bar, the frame's native height cannot be told exactly before the frame has been actually drawn. This means that in general you cannot use the native size to specify the initial size of a frame. As soon as you know the native size of a visible frame, you can calculate its outer size (see Section 30.3.1 [Frame Layout], page 793) by adding in the remaining components from the return value of `frame-geometry`. For invisible frames or for frames that have yet to be created, however, the outer size can only be estimated. This also means that calculating an exact initial position of a frame specified via offsets from the right or bottom edge of the screen (see Section 30.3.3 [Frame Position], page 800) is impossible.

The text size of any frame can be set and retrieved with the help of the `height` and `width` frame parameters (see Section 30.4.3.3 [Size Parameters], page 810). The text size of the initial frame can be also set with the help of an X-style geometry specification. See Section "Command Line Arguments for Emacs Invocation" in *The GNU Emacs Manual*. Below we list some functions to access and set the size of an existing, visible frame, by default the selected one.

`frame-height` **&optional** *frame* [Function]
`frame-width` **&optional** *frame* [Function]
: These functions return the height and width of the text area of *frame*, measured in units of the default font height and width of *frame* (see Section 30.3.2 [Frame Font], page 799). These functions are plain shorthands for writing `(frame-parameter frame 'height)` and `(frame-parameter frame 'width)`.

  If the text area of *frame* measured in pixels is not a multiple of its default font size, the values returned by these functions are rounded down to the number of characters of the default font that fully fit into the text area.

The functions following next return the pixel widths and heights of the native, outer and inner frame and the text area (see Section 30.3.1 [Frame Layout], page 793) of a given frame. For a text terminal, the results are in characters rather than pixels.

`frame-outer-width` **&optional** *frame* [Function]
`frame-outer-height` **&optional** *frame* [Function]
: These functions return the outer width and height of *frame* in pixels.

`frame-native-height` **&optional** *frame* [Function]
`frame-native-width` **&optional** *frame* [Function]
: These functions return the native width and height of *frame* in pixels.

`frame-inner-width` **&optional** *frame* [Function]
`frame-inner-height` **&optional** *frame* [Function]
: These functions return the inner width and height of *frame* in pixels.

`frame-text-width` **&optional** *frame* [Function]
`frame-text-height` **&optional** *frame* [Function]
These functions return the width and height of the text area of *frame* in pixels.

On window systems that support it, Emacs tries by default to make the text size of a frame measured in pixels a multiple of the frame's character size. This, however, usually means that a frame can be resized only in character size increments when dragging its external borders. It also may break attempts to truly maximize the frame or making it "fullheight" or "fullwidth" (see Section 30.4.3.3 [Size Parameters], page 810) leaving some empty space below and/or on the right of the frame. The following option may help in that case.

`frame-resize-pixelwise` [User Option]
If this option is `nil` (the default), a frame's text pixel size is usually rounded to a multiple of the current values of that frame's `frame-char-height` and `frame-char-width` whenever the frame is resized. If this is non-`nil`, no rounding occurs, hence frame sizes can increase/decrease by one pixel.

Setting this variable usually causes the next resize operation to pass the corresponding size hints to the window manager. This means that this variable should be set only in a user's initial file; applications should never bind it temporarily.

The precise meaning of a value of `nil` for this option depends on the toolkit used. Dragging the external border with the mouse is done character-wise provided the window manager is willing to process the corresponding size hints. Calling `set-frame-size` (see below) with arguments that do not specify the frame size as an integer multiple of its character size, however, may: be ignored, cause a rounding (GTK+), or be accepted (Lucid, Motif, MS-Windows).

With some window managers you may have to set this to non-`nil` in order to make a frame appear truly maximized or full-screen.

`set-frame-size` *frame width height* **&optional** *pixelwise* [Function]
This function sets the size of the text area of *frame*, measured in terms of the canonical height and width of a character on *frame* (see Section 30.3.2 [Frame Font], page 799).

The optional argument *pixelwise* non-`nil` means to measure the new width and height in units of pixels instead. Note that if `frame-resize-pixelwise` is `nil`, some toolkits may refuse to truly honor the request if it does not increase/decrease the frame size to a multiple of its character size.

`set-frame-height` *frame height* **&optional** *pretend pixelwise* [Function]
This function resizes the text area of *frame* to a height of *height* lines. The sizes of existing windows in *frame* are altered proportionally to fit.

If *pretend* is non-`nil`, then Emacs displays *height* lines of output in *frame*, but does not change its value for the actual height of the frame. This is only useful on text terminals. Using a smaller height than the terminal actually implements may be useful to reproduce behavior observed on a smaller screen, or if the terminal malfunctions when using its whole screen. Setting the frame height directly does not always work, because knowing the correct actual size may be necessary for correct cursor positioning on text terminals.

The optional fourth argument *pixelwise* non-`nil` means that *frame* should be *height* pixels high. Note that if `frame-resize-pixelwise` is `nil`, some window managers may refuse to truly honor the request if it does not increase/decrease the frame height to a multiple of its character height.

When used interactively, this command will ask the user for the number of lines to set the height of the currently selected frame. You can also provide this value with a numeric prefix.

`set-frame-width` *frame width* **&optional** *pretend pixelwise* [Function]
This function sets the width of the text area of *frame*, measured in characters. The argument *pretend* has the same meaning as in `set-frame-height`.

The optional fourth argument *pixelwise* non-`nil` means that *frame* should be *width* pixels wide. Note that if `frame-resize-pixelwise` is `nil`, some window managers may refuse to fully honor the request if it does not increase/decrease the frame width to a multiple of its character width.

When used interactively, this command will ask the user for the number of columns to set the width of the currently selected frame. You can also provide this value with a numeric prefix.

None of these three functions will make a frame smaller than needed to display all of its windows together with their scroll bars, fringes, margins, dividers, mode and header lines. This contrasts with requests by the window manager triggered, for example, by dragging the external border of a frame with the mouse. Such requests are always honored by clipping, if necessary, portions that cannot be displayed at the right, bottom corner of the frame. The parameters `min-width` and `min-height` (see Section 30.4.3.3 [Size Parameters], page 810) can be used to obtain a similar behavior when changing the frame size from within Emacs.

The abnormal hook `window-size-change-functions` (see Section 29.28 [Window Hooks], page 782) tracks all changes of the inner size of a frame including those induced by request of the window-system or window manager. To rule out false positives that might occur when changing only the sizes of a frame's windows without actually changing the size of the inner frame, use the following function.

`frame-size-changed-p` **&optional** *frame* [Function]
This function returns non-`nil` when the inner width or height of *frame* has changed since `window-size-change-functions` was run the last time for *frame*. It always returns `nil` immediately after running `window-size-change-functions` for *frame*.

30.3.5 Implied Frame Resizing

By default, Emacs tries to keep the number of lines and columns of a frame's text area unaltered when, for example, toggling its menu or tool bar, changing its default font or setting the width of any of its scroll bars. This means that in such case Emacs must ask the window manager to resize the frame's window in order to accommodate the size change.

Occasionally, such *implied frame resizing* may be unwanted, for example, when a frame has been maximized or made full-screen (where it's turned off by default). In general, users can disable implied resizing with the following option:

`frame-inhibit-implied-resize` [User Option]

If this option is `nil`, changing a frame's font, menu bar, tool bar, internal borders, fringes or scroll bars may resize its outer frame in order to keep the number of columns or lines of its text area unaltered. If this option is `t`, no such resizing is done.

The value of this option can be also a list of frame parameters. In that case, implied resizing is inhibited for the change of a parameter that appears in this list. Parameters currently handled by this option are `font`, `font-backend`, `internal-border-width`, `menu-bar-lines` and `tool-bar-lines`.

Changing any of the `scroll-bar-width`, `scroll-bar-height`, `vertical-scroll-bars`, `horizontal-scroll-bars`, `left-fringe` and `right-fringe` frame parameters is handled as if the frame contained just one live window. This means, for example, that removing vertical scroll bars on a frame containing several side by side windows will shrink the outer frame width by the width of one scroll bar provided this option is `nil` and keep it unchanged if this option is `t` or a list containing `vertical-scroll-bars`.

The default value is `(tab-bar-lines tool-bar-lines)` for Lucid, Motif and MS-Windows (which means that adding/removing a tool or tab bar there does not change the outer frame height), `(tab-bar-lines)` on all other window systems including GTK+ (which means that changing any of the parameters listed above with the exception of `tab-bar-lines` may change the size of the outer frame), and `t` otherwise (which means the outer frame size never changes implicitly when there's no window system support).

Note that when a frame is not large enough to accommodate a change of any of the parameters listed above, Emacs may try to enlarge the frame even if this option is non-`nil`.

Note also that window managers usually do not ask for resizing a frame when they change the number of lines occupied by an external menu or tool bar. Typically, such "wrappings" occur when a user shrinks a frame horizontally, making it impossible to display all elements of its menu or tool bar. They may also result from a change of the major mode altering the number of items of a menu or tool bar. Any such wrappings may implicitly alter the number of lines of a frame's text area and are unaffected by the setting of this option.

30.4 Frame Parameters

A frame has many parameters that control its appearance and behavior. Just what parameters a frame has depends on what display mechanism it uses.

Frame parameters exist mostly for the sake of graphical displays. Most frame parameters have no effect when applied to a frame on a text terminal; only the `height`, `width`, `name`, `title`, `menu-bar-lines`, `buffer-list` and `buffer-predicate` parameters do something special. If the terminal supports colors, the parameters `foreground-color`, `background-color`, `background-mode` and `display-type` are also meaningful. If the terminal supports frame transparency, the parameter `alpha` is also meaningful.

By default, frame parameters are saved and restored by the desktop library functions (see Section 24.8 [Desktop Save Mode], page 595) when the variable `desktop-restore-frames` is non-`nil`. It's the responsibility of applications that their parameters are included in

`frameset-persistent-filter-alist` to avoid that they get meaningless or even harmful values in restored sessions.

30.4.1 Access to Frame Parameters

These functions let you read and change the parameter values of a frame.

`frame-parameter` *frame parameter* [Function]
: This function returns the value of the parameter *parameter* (a symbol) of *frame*. If *frame* is `nil`, it returns the selected frame's parameter. If *frame* has no setting for *parameter*, this function returns `nil`.

`frame-parameters` **&optional** *frame* [Function]
: The function `frame-parameters` returns an alist listing all the parameters of *frame* and their values. If *frame* is `nil` or omitted, this returns the selected frame's parameters

`modify-frame-parameters` *frame alist* [Function]
: This function alters the frame *frame* based on the elements of *alist*. Each element of *alist* has the form `(parm . value)`, where *parm* is a symbol naming a parameter. If you don't mention a parameter in *alist*, its value doesn't change. If *frame* is `nil`, it defaults to the selected frame.

  Some parameters are only meaningful for frames on certain kinds of display (see Chapter 30 [Frames], page 787). If *alist* includes parameters that are not meaningful for the *frame*'s display, this function will change its value in the frame's parameter list, but will otherwise ignore it.

  When *alist* specifies more than one parameter whose value can affect the new size of *frame*, the final size of the frame may differ according to the toolkit used. For example, specifying that a frame should from now on have a menu and/or tool bar instead of none and simultaneously specifying the new height of the frame will inevitably lead to a recalculation of the frame's height. Conceptually, in such case, this function will try to have the explicit height specification prevail. It cannot be excluded, however, that the addition (or removal) of the menu or tool bar, when eventually performed by the toolkit, will defeat this intention.

  Sometimes, binding `frame-inhibit-implied-resize` (see Section 30.3.5 [Implied Frame Resizing], page 803) to a non-`nil` value around calls to this function may fix the problem sketched here. Sometimes, however, exactly such binding may be hit by the problem.

`set-frame-parameter` *frame parm value* [Function]
: This function sets the frame parameter *parm* to the specified *value*. If *frame* is `nil`, it defaults to the selected frame.

`modify-all-frames-parameters` *alist* [Function]
: This function alters the frame parameters of all existing frames according to *alist*, then modifies `default-frame-alist` (and, if necessary, `initial-frame-alist`) to apply the same parameter values to frames that will be created henceforth.

30.4.2 Initial Frame Parameters

You can specify the parameters for the initial startup frame by setting `initial-frame-alist` in your init file (see Section 42.1.2 [Init File], page 1263).

`initial-frame-alist` [User Option]

This variable's value is an alist of parameter values used when creating the initial frame. You can set this variable to specify the appearance of the initial frame without altering subsequent frames. Each element has the form:

```
(parameter . value)
```

Emacs creates the initial frame before it reads your init file. After reading that file, Emacs checks `initial-frame-alist`, and applies the parameter settings in the altered value to the already created initial frame.

If these settings affect the frame geometry and appearance, you'll see the frame appear with the wrong ones and then change to the specified ones. If that bothers you, you can specify the same geometry and appearance with X resources; those do take effect before the frame is created. See Section "X Resources" in *The GNU Emacs Manual*.

X resource settings typically apply to all frames. If you want to specify some X resources solely for the sake of the initial frame, and you don't want them to apply to subsequent frames, here's how to achieve this. Specify parameters in `default-frame-alist` to override the X resources for subsequent frames; then, to prevent these from affecting the initial frame, specify the same parameters in `initial-frame-alist` with values that match the X resources.

If these parameters include `(minibuffer . nil)`, that indicates that the initial frame should have no minibuffer. In this case, Emacs creates a separate *minibuffer-only frame* as well.

`minibuffer-frame-alist` [User Option]

This variable's value is an alist of parameter values used when creating an initial minibuffer-only frame (i.e., the minibuffer-only frame that Emacs creates if `initial-frame-alist` specifies a frame with no minibuffer).

`default-frame-alist` [User Option]

This is an alist specifying default values of frame parameters for all Emacs frames—the first frame, and subsequent frames. When using the X Window System, you can get the same results by means of X resources in many cases.

Setting this variable does not affect existing frames. Furthermore, functions that display a buffer in a separate frame may override the default parameters by supplying their own parameters.

If you invoke Emacs with command-line options that specify frame appearance, those options take effect by adding elements to either `initial-frame-alist` or `default-frame-alist`. Options which affect just the initial frame, such as '`--geometry`' and '`--maximized`', add to `initial-frame-alist`; the others add to `default-frame-alist`. see Section "Command Line Arguments for Emacs Invocation" in *The GNU Emacs Manual*.

30.4.3 Window Frame Parameters

Just what parameters a frame has depends on what display mechanism it uses. This section describes the parameters that have special meanings on some or all kinds of terminals. Of these, `name`, `title`, `height`, `width`, `buffer-list` and `buffer-predicate` provide meaningful information in terminal frames, and `tty-color-mode` is meaningful only for frames on text terminals.

30.4.3.1 Basic Parameters

These frame parameters give the most basic information about the frame. `title` and `name` are meaningful on all terminals.

`display` The display on which to open this frame. It should be a string of the form '*`host:dpy.screen`*', just like the `DISPLAY` environment variable. See Section 30.2 [Multiple Terminals], page 789, for more details about display names.

`display-type`
This parameter describes the range of possible colors that can be used in this frame. Its value is `color`, `grayscale` or `mono`.

`title` If a frame has a non-`nil` title, that title appears in the window system's title bar at the top of the frame, and also in the mode line of windows in that frame if `mode-line-frame-identification` uses '`%F`' (see Section 24.4.5 [%-Constructs], page 557). This is normally the case when Emacs is not using a window system, and can only display one frame at a time. When Emacs is using a window system, this parameter, if non-`nil`, overrides the title determined by the `name` parameter and the implicit title calculated according to `frame-title-format`. It also overrides the title determined by `icon-title-format` for iconified frames. See Section 30.6 [Frame Titles], page 823.

`name` The name of the frame. If you don't specify a name via this parameter, Emacs sets the frame name automatically, as specified by `frame-title-format` and `icon-title-format`, and that is the frame's title that will appear on display when Emacs uses a window system (unless the `title` parameter overrides it).

If you specify the frame name explicitly when you create the frame, the name is also used (instead of the name of the Emacs executable) when looking up X resources for the frame.

`explicit-name`
If the frame name was specified explicitly when the frame was created, this parameter will be that name. If the frame wasn't explicitly named, this parameter will be `nil`.

30.4.3.2 Position Parameters

Parameters describing the X- and Y-offsets of a frame are always measured in pixels. For a normal, non-child frame they specify the frame's outer position (see Section 30.3 [Frame Geometry], page 793) relative to its display's origin. For a child frame (see Section 30.14 [Child Frames], page 833) they specify the frame's outer position relative to the native

position of the frame's parent frame. (Note that none of these parameters is meaningful on TTY frames.)

`left` The position, in pixels, of the left outer edge of the frame with respect to the left edge of the frame's display or parent frame. It can be specified in one of the following ways.

an integer A positive integer always relates the left edge of the frame to the left edge of its display or parent frame. A negative integer relates the right frame edge to the right edge of the display or parent frame.

`(+` *pos*`)` This specifies the position of the left frame edge relative to the left edge of its display or parent frame. The integer *pos* may be positive or negative; a negative value specifies a position outside the screen or parent frame or on a monitor other than the primary one (for multi-monitor displays).

`(-` *pos*`)` This specifies the position of the right frame edge relative to the right edge of the display or parent frame. The integer *pos* may be positive or negative; a negative value specifies a position outside the screen or parent frame or on a monitor other than the primary one (for multi-monitor displays).

a floating-point value

A floating-point value in the range 0.0 to 1.0 specifies the left edge's offset via the *left position ratio* of the frame—the ratio of the left edge of its outer frame to the width of the frame's workarea (see Section 30.2 [Multiple Terminals], page 789) or its parent's native frame (see Section 30.14 [Child Frames], page 833) minus the width of the outer frame. Thus, a left position ratio of 0.0 flushes a frame to the left, a ratio of 0.5 centers it and a ratio of 1.0 flushes it to the right of its display or parent frame. Similarly, the *top position ratio* of a frame is the ratio of the frame's top position to the height of its workarea or parent frame minus the height of the frame.

Emacs will try to keep the position ratios of a child frame unaltered if that frame has a non-`nil` `keep-ratio` parameter (see Section 30.4.3.6 [Frame Interaction Parameters], page 814) and its parent frame is resized.

Since the outer size of a frame (see Section 30.3 [Frame Geometry], page 793) is usually unavailable before a frame has been made visible, it is generally not advisable to use floating-point values when creating decorated frames. Floating-point values are more suited for ensuring that an (undecorated) child frame is positioned nicely within the area of its parent frame.

Some window managers ignore program-specified positions. If you want to be sure the position you specify is not ignored, specify a non-`nil` value for the `user-position` parameter as in the following example:

```
(modify-frame-parameters
  nil '((user-position . t) (left . (+ -4))))
```

In general, it is not a good idea to position a frame relative to the right or bottom edge of its display. Positioning the initial or a new frame is either not accurate (because the size of the outer frame is not yet fully known before the frame has been made visible) or will cause additional flicker (if the frame has to be repositioned after becoming visible).

Note also, that positions specified relative to the right/bottom edge of a display, workarea or parent frame as well as floating-point offsets are stored internally as integer offsets relative to the left/top edge of the display, workarea or parent frame edge. They are also returned as such by functions like `frame-parameters` and restored as such by the desktop saving routines.

`top` The screen position of the top (or bottom) edge, in pixels, with respect to the top (or bottom) edge of the display or parent frame. It works just like `left`, except vertically instead of horizontally.

`icon-left`
The screen position of the left edge of the frame's icon, in pixels, counting from the left edge of the screen. This takes effect when the frame is iconified, if the window manager supports this feature. If you specify a value for this parameter, then you must also specify a value for `icon-top` and vice versa.

`icon-top` The screen position of the top edge of the frame's icon, in pixels, counting from the top edge of the screen. This takes effect when the frame is iconified, if the window manager supports this feature.

`user-position`
When you create a frame and specify its screen position with the `left` and `top` parameters, use this parameter to say whether the specified position was user-specified (explicitly requested in some way by a human user) or merely program-specified (chosen by a program). A non-`nil` value says the position was user-specified.

Window managers generally heed user-specified positions, and some heed program-specified positions too. But many ignore program-specified positions, placing the window in a default fashion or letting the user place it with the mouse. Some window managers, including `twm`, let the user specify whether to obey program-specified positions or ignore them.

When you call `make-frame`, you should specify a non-`nil` value for this parameter if the values of the `left` and `top` parameters represent the user's stated preference; otherwise, use `nil`.

`z-group` This parameter specifies a relative position of the frame's window-system window in the stacking (Z-) order of the frame's display.

If this is `above`, the window-system will display the window that corresponds to the frame above all other window-system windows that do not have the `above` property set. If this is `nil`, the frame's window is displayed below all windows that have the `above` property set and above all windows that have the `below` property set. If this is `below`, the frame's window is displayed below all windows that do not have the `below` property set.

To position the frame above or below a specific other frame use the function `frame-restack` (see Section 30.12 [Raising and Lowering], page 832).

30.4.3.3 Size Parameters

Frame parameters usually specify frame sizes in character units. On graphical displays, the `default` face determines the actual pixel sizes of these character units (see Section 41.12.1 [Face Attributes], page 1168).

`width` This parameter specifies the width of the frame. It can be specified as in the following ways:

an integer A positive integer specifies the width of the frame's text area (see Section 30.3 [Frame Geometry], page 793) in characters.

a cons cell If this is a cons cell with the symbol `text-pixels` in its CAR, the CDR of that cell specifies the width of the frame's text area in pixels.

a floating-point value
A floating-point number between 0.0 and 1.0 can be used to specify the width of a frame via its *width ratio*—the ratio of its outer width (see Section 30.3 [Frame Geometry], page 793) to the width of the frame's workarea (see Section 30.2 [Multiple Terminals], page 789) or its parent frame's (see Section 30.14 [Child Frames], page 833) native frame. Thus, a value of 0.5 makes the frame occupy half of the width of its workarea or parent frame, a value of 1.0 the full width. Similarly, the *height ratio* of a frame is the ratio of its outer height to the height of its workarea or its parent's native frame.

Emacs will try to keep the width and height ratio of a child frame unaltered if that frame has a non-`nil` `keep-ratio` parameter (see Section 30.4.3.6 [Frame Interaction Parameters], page 814) and its parent frame is resized.

Since the outer size of a frame is usually unavailable before a frame has been made visible, it is generally not advisable to use floating-point values when creating decorated frames. Floating-point values are more suited to ensure that a child frame always fits within the area of its parent frame as, for example, when customizing `display-buffer-alist` (see Section 29.13.1 [Choosing Window], page 726) via `display-buffer-in-child-frame`.

Regardless of how this parameter was specified, functions reporting the value of this parameter like `frame-parameters` always report the width of the frame's text area in characters as an integer rounded, if necessary, to a multiple of the frame's default character width. That value is also used by the desktop saving routines.

`height` This parameter specifies the height of the frame. It works just like `width`, except vertically instead of horizontally.

`user-size`
: This does for the size parameters `height` and `width` what the `user-position` parameter (see Section 30.4.3.2 [Position Parameters], page 807) does for the position parameters `top` and `left`.

`min-width`
: This parameter specifies the minimum native width (see Section 30.3 [Frame Geometry], page 793) of the frame, in characters. Normally, the functions that establish a frame's initial width or resize a frame horizontally make sure that all the frame's windows, vertical scroll bars, fringes, margins and vertical dividers can be displayed. This parameter, if non-`nil` allows to make a frame narrower than that with the consequence that any components that do not fit will be clipped by the window manager.

`min-height`
: This parameter specifies the minimum native height (see Section 30.3 [Frame Geometry], page 793) of the frame, in characters. Normally, the functions that establish a frame's initial size or resize a frame make sure that all the frame's windows, horizontal scroll bars and dividers, mode and header lines, the echo area and the internal menu and tool bar can be displayed. This parameter, if non-`nil` allows to make a frame smaller than that with the consequence that any components that do not fit will be clipped by the window manager.

`fullscreen`
: This parameter specifies whether to maximize the frame's width, height or both. Its value can be `fullwidth`, `fullheight`, `fullboth`, or `maximized`.[1] A *fullwidth* frame is as wide as possible, a *fullheight* frame is as tall as possible, and a *fullboth* frame is both as wide and as tall as possible. A *maximized* frame is like a "fullboth" frame, except that it usually keeps its title bar and the buttons for resizing and closing the frame. Also, maximized frames typically avoid hiding any task bar or panels displayed on the desktop. A "fullboth" frame, on the other hand, usually omits the title bar and occupies the entire available screen space.

  Full-height and full-width frames are more similar to maximized frames in this regard. However, these typically display an external border which might be absent with maximized frames. Hence the heights of maximized and full-height frames and the widths of maximized and full-width frames often differ by a few pixels.

  With some window managers you may have to customize the variable `frame-resize-pixelwise` (see Section 30.3.4 [Frame Size], page 801) in order to make a frame truly appear maximized or full-screen. Moreover, some window managers might not support smooth transition between the various full-screen or maximization states. Customizing the variable `x-frame-normalize-before-maximize` can help to overcome that.

  Full-screen on macOS hides both the tool-bar and the menu-bar, however both will be displayed if the mouse pointer is moved to the top of the screen.

[1] On PGTK frames, setting the values `fullheight` and `fullwidth` has no effect.

`fullscreen-restore`
: This parameter specifies the desired fullscreen state of the frame after invoking the `toggle-frame-fullscreen` command (see Section "Frame Commands" in *The GNU Emacs Manual*) in the "fullboth" state. Normally this parameter is installed automatically by that command when toggling the state to fullboth. If, however, you start Emacs in the "fullboth" state, you have to specify the desired behavior in your initial file as, for example

```
(setq default-frame-alist
    '((fullscreen . fullboth)
      (fullscreen-restore . fullheight)))
```

This will give a new frame full height after typing in it `F11` for the first time.

`fit-frame-to-buffer-margins`
: This parameter allows to override the value of the option `fit-frame-to-buffer-margins` when fitting this frame to the buffer of its root window with `fit-frame-to-buffer` (see Section 29.5 [Resizing Windows], page 705).

`fit-frame-to-buffer-sizes`
: This parameter allows to override the value of the option `fit-frame-to-buffer-sizes` when fitting this frame to the buffer of its root window with `fit-frame-to-buffer` (see Section 29.5 [Resizing Windows], page 705).

30.4.3.4 Layout Parameters

These frame parameters enable or disable various parts of the frame, or control their sizes.

`border-width`
: The width in pixels of the frame's outer border (see Section 30.3 [Frame Geometry], page 793).

`internal-border-width`
: The width in pixels of the frame's internal border (see Section 30.3 [Frame Geometry], page 793).

`child-frame-border-width`
: The width in pixels of the frame's internal border (see Section 30.3 [Frame Geometry], page 793) if the given frame is a child frame (see Section 30.14 [Child Frames], page 833). If this is `nil`, the value specified by the `internal-border-width` parameter is used instead.

`vertical-scroll-bars`
: Whether the frame has scroll bars (see Section 41.14 [Scroll Bars], page 1199) for vertical scrolling, and which side of the frame they should be on. The possible values are `left`, `right`, and `nil` for no scroll bars.

`horizontal-scroll-bars`
: Whether the frame has scroll bars for horizontal scrolling (`t` and `bottom` mean yes, `nil` means no).

`scroll-bar-width`
: The width of vertical scroll bars, in pixels, or `nil` meaning to use the default width.

`scroll-bar-height`
: The height of horizontal scroll bars, in pixels, or `nil` meaning to use the default height.

`left-fringe`
`right-fringe`
: The default width of the left and right fringes of windows in this frame (see Section 41.13 [Fringes], page 1193). If either of these is zero, that effectively removes the corresponding fringe.

  When you use `frame-parameter` to query the value of either of these two frame parameters, the return value is always an integer. When using `set-frame-parameter`, passing a `nil` value imposes an actual default value of 8 pixels.

`right-divider-width`
: The width (thickness) reserved for the right divider (see Section 41.15 [Window Dividers], page 1202) of any window on the frame, in pixels. A value of zero means to not draw right dividers.

`bottom-divider-width`
: The width (thickness) reserved for the bottom divider (see Section 41.15 [Window Dividers], page 1202) of any window on the frame, in pixels. A value of zero means to not draw bottom dividers.

`menu-bar-lines`
: The number of lines to allocate at the top of the frame for a menu bar (see Section 23.18.5 [Menu Bar], page 517). The default is one if Menu Bar mode is enabled and zero otherwise. See Section "Menu Bars" in *The GNU Emacs Manual*. For an external menu bar (see Section 30.3.1 [Frame Layout], page 793), this value remains unchanged even when the menu bar wraps to two or more lines. In that case, the `menu-bar-size` value returned by `frame-geometry` (see Section 30.3 [Frame Geometry], page 793) allows to derive whether the menu bar actually occupies one or more lines.

`tool-bar-lines`
: The number of lines to use for the tool bar (see Section 23.18.6 [Tool Bar], page 518). The default is one if Tool Bar mode is enabled and zero otherwise. See Section "Tool Bars" in *The GNU Emacs Manual*. This value may change whenever the tool bar wraps (see Section 30.3.1 [Frame Layout], page 793).

`tool-bar-position`
: The position of the tool bar when Emacs was built with GTK+. Its value can be one of `top`, `bottom` `left`, `right`. The default is `top`.

`tab-bar-lines`
: The number of lines to use for the tab bar (see Section "Tab Bars" in *The GNU Emacs Manual*). The default is one if Tab Bar mode is enabled and zero otherwise. This value may change whenever the tab bar wraps (see Section 30.3.1 [Frame Layout], page 793).

`line-spacing`
: Additional space to leave below each text line, in pixels (a positive integer). See Section 41.11 [Line Height], page 1166, for more information.

`no-special-glyphs`
: If this is non-`nil`, it suppresses the display of any truncation (see Section 41.3 [Truncation], page 1134) and continuation glyphs for all the buffers displayed by this frame. This is useful to eliminate such glyphs when fitting a frame to its buffer via `fit-frame-to-buffer` (see Section 29.5 [Resizing Windows], page 705). This frame parameter has effect only for GUI frames shown on graphical displays, and only if the fringes are disabled. This parameter is intended as a purely-presentation feature, and in particular should not be used for frames where the user can interactively insert text, or more generally where the cursor is shown. A notable example of frames where this is used is tooltip frames (see Section 41.26 [Tooltips], page 1254).

30.4.3.5 Buffer Parameters

These frame parameters, meaningful on all kinds of terminals, deal with which buffers have been, or should, be displayed in the frame.

`minibuffer`
: Whether this frame has its own minibuffer. The value `t` means yes, `nil` means no, `only` means this frame is just a minibuffer. If the value is a minibuffer window (in some other frame), the frame uses that minibuffer.

  This parameter takes effect when the frame is created. If specified as `nil`, Emacs will try to set it to the minibuffer window of `default-minibuffer-frame` (see Section 30.9 [Minibuffers and Frames], page 826). For an existing frame, this parameter can be used exclusively to specify another minibuffer window. It is not allowed to change it from a minibuffer window to `t` and vice-versa, or from `t` to `nil`. If the parameter specifies a minibuffer window already, setting it to `nil` has no effect.

  The special value `child-frame` means to make a minibuffer-only child frame (see Section 30.14 [Child Frames], page 833) whose parent becomes the frame created. As if specified as `nil`, Emacs will set this parameter to the minibuffer window of the child frame but will not select the child frame after its creation.

`buffer-predicate`
: The buffer-predicate function for this frame. The function `other-buffer` uses this predicate (from the selected frame) to decide which buffers it should consider, if the predicate is not `nil`. It calls the predicate with one argument, a buffer, once for each buffer; if the predicate returns a non-`nil` value, it considers that buffer.

`buffer-list`
: A list of buffers that have been selected in this frame, ordered most-recently-selected first.

`unsplittable`
: If non-`nil`, this frame's window is never split automatically.

30.4.3.6 Frame Interaction Parameters

These parameters supply forms of interactions between different frames.

`parent-frame`
: If non-`nil`, this means that this frame is a child frame (see Section 30.14 [Child Frames], page 833), and this parameter specifies its parent frame. If `nil`, this means that this frame is a normal, top-level frame.

`delete-before`
: If non-`nil`, this parameter specifies another frame whose deletion will automatically trigger the deletion of this frame. See Section 30.7 [Deleting Frames], page 824.

`mouse-wheel-frame`
: If non-`nil`, this parameter specifies the frame whose windows will be scrolled whenever the mouse wheel is scrolled with the mouse pointer hovering over this frame, see Section "Mouse Commands" in *The GNU Emacs Manual*.

`no-other-frame`
: If this is non-`nil`, then this frame is not eligible as candidate for the functions `next-frame`, `previous-frame` (see Section 30.8 [Finding All Frames], page 825) and `other-frame`, see Section "Frame Commands" in *The GNU Emacs Manual*.

`auto-hide-function`
: When this parameter specifies a function, that function will be called instead of the function specified by the variable `frame-auto-hide-function` when quitting the frame's only window (see Section 29.16 [Quitting Windows], page 752) and there are other frames left.

`minibuffer-exit`
: When this parameter is non-`nil`, Emacs will by default make this frame invisible whenever the minibuffer (see Chapter 21 [Minibuffers], page 386) is exited. Alternatively, it can specify the functions `iconify-frame` and `delete-frame`. This parameter is useful to make a child frame disappear automatically (similar to how Emacs deals with a window) when exiting the minibuffer.

`keep-ratio`
: This parameter is currently meaningful for child frames (see Section 30.14 [Child Frames], page 833) only. If it is non-`nil`, then Emacs will try to keep the frame's size (width and height) ratios (see Section 30.4.3.3 [Size Parameters], page 810) as well as its left and right position ratios (see Section 30.4.3.2 [Position Parameters], page 807) unaltered whenever its parent frame is resized.

  If the value of this parameter is `nil`, the frame's position and size remain unaltered when the parent frame is resized, so the position and size ratios may change. If the value of this parameter is `t`, Emacs will try to preserve the frame's size and position ratios, hence the frame's size and position relative to its parent frame may change.

  More individual control is possible by using a cons cell: In that case the frame's width ratio is preserved if the CAR of the cell is either `t` or `width-only`. The height ratio is preserved if the CAR of the cell is either `t` or `height-only`. The left position ratio is preserved if the CDR of the cell is either `t` or `left-only`. The top position ratio is preserved if the CDR of the cell is either `t` or `top-only`.

30.4.3.7 Mouse Dragging Parameters

The parameters described below provide support for resizing a frame by dragging its internal borders with the mouse. They also allow moving a frame with the mouse by dragging the header or tab line of its topmost or the mode line of its bottommost window.

These parameters are mostly useful for child frames (see Section 30.14 [Child Frames], page 833) that come without window manager decorations. If necessary, they can be used for undecorated top-level frames as well.

`drag-internal-border`
: If non-`nil`, the frame can be resized by dragging its internal borders, if present, with the mouse.

`drag-with-header-line`
: If non-`nil`, the frame can be moved with the mouse by dragging the header line of its topmost window.

`drag-with-tab-line`
: If non-`nil`, the frame can be moved with the mouse by dragging the tab line of its topmost window.

`drag-with-mode-line`
: If non-`nil`, the frame can be moved with the mouse by dragging the mode line of its bottommost window. Note that such a frame is not allowed to have its own minibuffer window.

`snap-width`
: A frame that is moved with the mouse will "snap" at the border(s) of the display or its parent frame whenever it is dragged as near to such an edge as the number of pixels specified by this parameter.

`top-visible`
: If this parameter is a number, the top edge of the frame never appears above the top edge of its display or parent frame. Moreover, as many pixels of the frame as specified by that number will remain visible when the frame is moved against any of the remaining edges of its display or parent frame. Setting this parameter is useful to guard against dragging a child frame with a non-`nil` `drag-with-header-line` parameter completely out of the area of its parent frame.

`bottom-visible`
: If this parameter is a number, the bottom edge of the frame never appears below the bottom edge of its display or parent frame. Moreover, as many pixels of the frame as specified by that number will remain visible when the frame is moved against any of the remaining edges of its display or parent frame. Setting this parameter is useful to guard against dragging a child frame with a non-`nil` `drag-with-mode-line` parameter completely out of the area of its parent frame.

30.4.3.8 Window Management Parameters

The following frame parameters control various aspects of the frame's interaction with the window manager or window system. They have no effect on text terminals.

`visibility`
: The state of visibility of the frame. There are three possibilities: `nil` for invisible, `t` for visible, and `icon` for iconified. See Section 30.11 [Visibility of Frames], page 831.

`auto-raise`
: If non-`nil`, Emacs automatically raises the frame when it is selected. Some window managers do not allow this.

`auto-lower`
: If non-`nil`, Emacs automatically lowers the frame when it is deselected. Some window managers do not allow this.

`icon-type`
: The type of icon to use for this frame. If the value is a string, that specifies a file containing a bitmap to use; `nil` specifies no icon (in which case the window manager decides what to show); any other non-`nil` value specifies the default Emacs icon.

`icon-name`
: The name to use in the icon for this frame, when and if the icon appears. If this is `nil`, the frame's title is used.

`window-id`
: The ID number which the graphical display uses for this frame. Emacs assigns this parameter when the frame is created; changing the parameter has no effect on the actual ID number.

`outer-window-id`
: The ID number of the outermost window-system window in which the frame exists. As with `window-id`, changing this parameter has no actual effect.

`wait-for-wm`
: If non-`nil`, tell Xt to wait for the window manager to confirm geometry changes. Some window managers, including versions of Fvwm2 and KDE, fail to confirm, so Xt hangs. Set this to `nil` to prevent hanging with those window managers.

`sticky`
: If non-`nil`, the frame is visible on all virtual desktops on systems with virtual desktops.

`shaded`
: If non-`nil`, tell the window manager to display the frame in a way that its contents are hidden, leaving only the title bar.

`use-frame-synchronization`
: If non-`nil`, synchronize the frame redisplay with the refresh rate of the monitor to avoid graphics tearing. At present, this is only implemented on Haiku and the X window system inside no-toolkit and X toolkit builds, does not work correctly with toolkit scroll bars, and requires a compositing manager supporting the relevant display synchronization protocols. The `synchronizeResize` X resource must also be set to the string `"extended"`.

`inhibit-double-buffering`
: If non-`nil`, the frame is drawn to the screen without double buffering. Emacs normally attempts to use double buffering, where available, to reduce flicker.

Set this property if you experience display bugs or pine for that retro, flicker-y feeling.

`skip-taskbar`
: If non-`nil`, this tells the window manager to remove the frame's icon from the taskbar associated with the frame's display and inhibit switching to the frame's window via the combination *`Alt`*`-TAB`. On MS-Windows, iconifying such a frame will "roll in" its window-system window at the bottom of the desktop. Some window managers may not honor this parameter.

`no-focus-on-map`
: If non-`nil`, this means that the frame does not want to receive input focus when it is mapped (see Section 30.11 [Visibility of Frames], page 831). Some window managers may not honor this parameter.

`no-accept-focus`
: If non-`nil`, this means that the frame does not want to receive input focus via explicit mouse clicks or when moving the mouse into it either via `focus-follows-mouse` (see Section 30.10 [Input Focus], page 827) or `mouse-autoselect-window` (see Section 29.25 [Mouse Window Auto-selection], page 776). This may have the unwanted side-effect that a user cannot scroll a non-selected frame with the mouse. Some window managers may not honor this parameter. On Haiku, it also has the side-effect that the window will not be able to receive any keyboard input from the user, not even if the user switches to the frame using the key combination *`Alt`*`-TAB`.

`undecorated`
: If non-`nil`, this frame's window-system window is drawn without decorations, like the title, minimize/maximize boxes and external borders. This usually means that the window cannot be dragged, resized, iconified, maximized or deleted with the mouse. If `nil`, the frame's window is usually drawn with all the elements listed above unless their display has been suspended via window manager settings.

  Under X, Emacs uses the Motif window manager hints to turn off decorations. Some window managers may not honor these hints.

  NS builds consider the tool bar to be a decoration, and therefore hide it on an undecorated frame.

`override-redirect`
: If non-`nil`, this means that this is an *override redirect* frame—a frame not handled by window managers under X. Override redirect frames have no window manager decorations, can be positioned and resized only via Emacs' positioning and resizing functions and are usually drawn on top of all other frames. Setting this parameter has no effect on MS-Windows.

`ns-appearance`
: Only available on macOS, if set to `dark` draw this frame's window-system window using the "vibrant dark" theme, and if set to `light` use the "aqua" theme, otherwise use the system default. The "vibrant dark" theme can be used to set

the toolbar and scrollbars to a dark appearance when using an Emacs theme with a dark background.

`ns-transparent-titlebar`
: Only available on macOS, if non-`nil`, set the titlebar and toolbar to be transparent. This effectively sets the background color of both to match the Emacs background color.

30.4.3.9 Cursor Parameters

This frame parameter controls the way the cursor looks.

`cursor-type`
: How to display the cursor. Legitimate values are:

  `box` Display a filled box. (This is the default.)

  `(box . size)`
  : Display a filled box. However, display it as a hollow box if point is under masked image larger than *size* pixels in either dimension.

  `hollow` Display a hollow box.

  `nil` Don't display a cursor.

  `bar` Display a vertical bar between characters.

  `(bar . width)`
  : Display a vertical bar *width* pixels wide between characters.

  `hbar` Display a horizontal bar.

  `(hbar . height)`
  : Display a horizontal bar *height* pixels high.

The `cursor-type` frame parameter may be overridden by the variables `cursor-type` and `cursor-in-non-selected-windows`:

`cursor-type` [User Option]
: This buffer-local variable controls how the cursor looks in a selected window showing the buffer. If its value is `t`, that means to use the cursor specified by the `cursor-type` frame parameter. Otherwise, the value should be one of the cursor types listed above, and it overrides the `cursor-type` frame parameter.

`cursor-in-non-selected-windows` [User Option]
: This buffer-local variable controls how the cursor looks in a window that is not selected. It supports the same values as the `cursor-type` frame parameter; also, `nil` means don't display a cursor in nonselected windows, and `t` (the default) means use a standard modification of the usual cursor type (solid box becomes hollow box, and bar becomes a narrower bar).

`x-stretch-cursor` [User Option]
: This variable controls the width of the block cursor displayed on extra-wide glyphs such as a tab or a stretch of white space. By default, the block cursor is only as wide as the font's default character, and will not cover all of the width of the glyph under

it if that glyph is extra-wide. A non-`nil` value of this variable means draw the block cursor as wide as the glyph under it. The default value is `nil`.

This variable has no effect on text-mode frames, since the text-mode cursor is drawn by the terminal out of Emacs's control.

`blink-cursor-alist` [User Option]

This variable specifies how to blink the cursor. Each element has the form `(`*`on-state . off-state`*`)`. Whenever the cursor type equals *on-state* (comparing using `equal`), the corresponding *off-state* specifies what the cursor looks like when it blinks off. Both *on-state* and *off-state* should be suitable values for the `cursor-type` frame parameter.

There are various defaults for how to blink each type of cursor, if the type is not mentioned as an *on-state* here. Changes in this variable do not take effect immediately, only when you specify the `cursor-type` frame parameter.

30.4.3.10 Font and Color Parameters

These frame parameters control the use of fonts and colors.

`font-backend`

A list of symbols, specifying the *font backends* to use for drawing characters on the frame, in order of priority. In Emacs built without Cairo drawing on X, there are currently three potentially available font backends: `x` (the X core font driver), `xft` (the Xft font driver), and `xfthb` (the Xft font driver with HarfBuzz text shaping). If built with Cairo drawing, there are also three potentially available font backends on X: `x`, `ftcr` (the FreeType font driver on Cairo), and `ftcrhb` (the FreeType font driver on Cairo with HarfBuzz text shaping). When Emacs is built with HarfBuzz, the default font driver is `ftcrhb`, although use of the `ftcr` driver is still possible, but not recommended. On MS-Windows, there are currently three available font backends: `gdi` (the core MS-Windows font driver), `uniscribe` (font driver for OTF and TTF fonts with text shaping by the Uniscribe engine), and `harfbuzz` (font driver for OTF and TTF fonts with HarfBuzz text shaping) (see Section "Windows Fonts" in *The GNU Emacs Manual*). The `harfbuzz` driver is similarly recommended. On Haiku, there can be several font drivers (see Section "Haiku Fonts" in *The GNU Emacs Manual*).

On other systems, there is only one available font backend, so it does not make sense to modify this frame parameter.

`background-mode`

This parameter is either `dark` or `light`, according to whether the background color is a light one or a dark one.

`tty-color-mode`

This parameter overrides the terminal's color support as given by the system's terminal capabilities database in that this parameter's value specifies the color mode to use on a text terminal. The value can be either a symbol or a number. A number specifies the number of colors to use (and, indirectly, what commands to issue to produce each color). For example, `(tty-color-mode . 8)` specifies

use of the ANSI escape sequences for 8 standard text colors. A value of −1 turns off color support.

If the parameter's value is a symbol, it specifies a number through the value of `tty-color-mode-alist`, and the associated number is used instead.

`screen-gamma`
: If this is a number, Emacs performs gamma correction which adjusts the brightness of all colors. The value should be the screen gamma of your display.

  Usual PC monitors have a screen gamma of 2.2, so color values in Emacs, and in X windows generally, are calibrated to display properly on a monitor with that gamma value. If you specify 2.2 for `screen-gamma`, that means no correction is needed. Other values request correction, designed to make the corrected colors appear on your screen the way they would have appeared without correction on an ordinary monitor with a gamma value of 2.2.

  If your monitor displays colors too light, you should specify a `screen-gamma` value smaller than 2.2. This requests correction that makes colors darker. A screen gamma value of 1.5 may give good results for LCD color displays.

`alpha`
: This parameter specifies the opacity of the frame, on graphical displays that support variable opacity. It should be an integer between 0 and 100, where 0 means completely transparent and 100 means completely opaque. It can also have a `nil` value, which tells Emacs not to set the frame opacity (leaving it to the window manager).

  To prevent the frame from disappearing completely from view, the variable `frame-alpha-lower-limit` defines a lower opacity limit. If the value of the frame parameter is less than the value of this variable, Emacs uses the latter. By default, `frame-alpha-lower-limit` is 20.

  The `alpha` frame parameter can also be a cons cell `(`*`active`*` . `*`inactive`*`)`, where *active* is the opacity of the frame when it is selected, and *inactive* is the opacity when it is not selected.

  Some window systems do not support the `alpha` parameter for child frames (see Section 30.14 [Child Frames], page 833).

`alpha-background`
: Sets the background transparency of the frame. Unlike the `alpha` frame parameter, this only controls the transparency of the background while keeping foreground elements such as text fully opaque. It should be an integer between 0 and 100, where 0 means completely transparent and 100 means completely opaque (default).

The following frame parameters are semi-obsolete in that they are automatically equivalent to particular face attributes of particular faces (see Section "Standard Faces" in *The Emacs Manual*):

`font`
: The name of the font for displaying text in the frame. This is a string, either a valid font name for your system or the name of an Emacs fontset (see Section 41.12.11 [Fontsets], page 1186). It is equivalent to the `font` attribute of the `default` face.

`foreground-color`
: The color to use for characters. It is equivalent to the `:foreground` attribute of the `default` face.

`background-color`
: The color to use for the background of characters. It is equivalent to the `:background` attribute of the `default` face.

`mouse-color`
: The color for the mouse pointer. It is equivalent to the `:background` attribute of the `mouse` face.

`cursor-color`
: The color for the cursor that shows point. It is equivalent to the `:background` attribute of the `cursor` face.

`border-color`
: The color for the border of the frame. It is equivalent to the `:background` attribute of the `border` face.

`scroll-bar-foreground`
: If non-`nil`, the color for the foreground of scroll bars. It is equivalent to the `:foreground` attribute of the `scroll-bar` face.

`scroll-bar-background`
: If non-`nil`, the color for the background of scroll bars. It is equivalent to the `:background` attribute of the `scroll-bar` face.

30.4.4 Geometry

Here's how to examine the data in an X-style window geometry specification:

`x-parse-geometry` *geom* [Function]
: The function `x-parse-geometry` converts a standard X window geometry string to an alist that you can use as part of the argument to `make-frame`.

  The alist describes which parameters were specified in *geom*, and gives the values specified for them. Each element looks like `(`*parameter*` . `*value*`)`. The possible *parameter* values are `left`, `top`, `width`, and `height`.

  For the size parameters, the value must be an integer. The position parameter names `left` and `top` are not totally accurate, because some values indicate the position of the right or bottom edges instead. The *value* possibilities for the position parameters are: an integer, a list `(+ `*pos*`)`, or a list `(- `*pos*`)`; as previously described (see Section 30.4.3.2 [Position Parameters], page 807).

  Here is an example:

```
(x-parse-geometry "35x70+0-0")
     ⇒ ((height . 70) (width . 35)
         (top - 0) (left . 0))
```

30.5 Terminal Parameters

Each terminal has a list of associated parameters. These *terminal parameters* are mostly a convenient way of storage for terminal-local variables, but some terminal parameters have a special meaning.

This section describes functions to read and change the parameter values of a terminal. They all accept as their argument either a terminal or a frame; the latter means use that frame's terminal. An argument of `nil` means the selected frame's terminal.

`terminal-parameters` **&optional** *terminal* [Function]
This function returns an alist listing all the parameters of *terminal* and their values.

`terminal-parameter` *terminal parameter* [Function]
This function returns the value of the parameter *parameter* (a symbol) of *terminal*. If *terminal* has no setting for *parameter*, this function returns `nil`.

`set-terminal-parameter` *terminal parameter value* [Function]
This function sets the parameter *parameter* of *terminal* to the specified *value*, and returns the previous value of that parameter.

Here's a list of a few terminal parameters that have a special meaning:

`background-mode`
The classification of the terminal's background color, either `light` or `dark`.

`normal-erase-is-backspace`
Value is either 1 or 0, depending on whether `normal-erase-is-backspace-mode` is turned on or off on this terminal. See Section "DEL Does Not Delete" in *The Emacs Manual*.

`terminal-initted`
After the terminal is initialized, this is set to the terminal-specific initialization function.

`tty-mode-set-strings`
When present, a list of strings containing escape sequences that Emacs will output while configuring a tty for rendering. Emacs emits these strings only when configuring a terminal: if you want to enable a mode on a terminal that is already active (for example, while in `tty-setup-hook`), explicitly output the necessary escape sequence using `send-string-to-terminal` in addition to adding the sequence to `tty-mode-set-strings`.

`tty-mode-reset-strings`
When present, a list of strings that undo the effects of the strings in `tty-mode-set-strings`. Emacs emits these strings when exiting, deleting a terminal, or suspending itself.

30.6 Frame Titles

Every frame has a `name` parameter; this serves as the default for the frame title which window systems typically display at the top of the frame. You can specify a name explicitly by setting the `name` frame property.

Normally you don't specify the name explicitly, and Emacs computes the frame name automatically based on a template stored in the variable `frame-title-format`. Emacs recomputes the name each time the frame is redisplayed.

`frame-title-format` [Variable]
: This variable specifies how to compute a name for a frame when you have not explicitly specified one (via the frame's parameters; see Section 30.4.3.1 [Basic Parameters], page 807). The variable's value is actually a mode line construct, just like `mode-line-format`, except that the '`%c`', '`%C`', and '`%l`' constructs are ignored. See Section 24.4.2 [Mode Line Data], page 552.

`icon-title-format` [Variable]
: This variable specifies how to compute the name for an iconified frame when you have not explicitly specified the frame's name via the frame's parameters. The resulting title appears in the frame's icon itself. If the value is a string, is should be a mode line construct like that of `frame-title-format`. The value can also be `t`, which means to use `frame-title-format` instead; this avoids problems with some window managers and desktop environments, where a change in a frame's title (when a frame is iconified) is interpreted as a request to raise the frame and/or give it input focus. It is also useful if you want the frame's title to be the same no matter if the frame is iconified or not. The default value is a string identical to the default value of `frame-title-format`.

`multiple-frames` [Variable]
: This variable is set automatically by Emacs. Its value is `t` when there are two or more frames (not counting minibuffer-only frames or invisible frames). The default value of `frame-title-format` uses `multiple-frames` so as to put the buffer name in the frame title only when there is more than one frame.

  The value of this variable is not guaranteed to be accurate except while processing `frame-title-format` or `icon-title-format`.

30.7 Deleting Frames

A *live frame* is one that has not been deleted. When a frame is deleted, it is removed from its terminal display, although it may continue to exist as a Lisp object until there are no more references to it.

`delete-frame` **&optional** *frame force* [Command]
: This function deletes the frame *frame*. The argument *frame* must specify a live frame (see below) and defaults to the selected frame.

  It first deletes any child frame of *frame* (see Section 30.14 [Child Frames], page 833) and any frame whose `delete-before` frame parameter (see Section 30.4.3.6 [Frame Interaction Parameters], page 814) specifies *frame*. All such deletions are performed recursively; so this step makes sure that no other frames with *frame* as their ancestor will exist. Then, unless *frame* specifies a tooltip, this function runs the hook `delete-frame-functions` (each function getting one argument, *frame*) before actually killing the frame. After actually killing the frame and removing the frame from the frame list, `delete-frame` runs `after-delete-frame-functions`.

Note that a frame cannot be deleted as long as its minibuffer serves as surrogate minibuffer for another frame (see Section 30.9 [Minibuffers and Frames], page 826). Normally, you cannot delete a frame if all other frames are invisible, but if *force* is non-`nil`, then you are allowed to do so.

`frame-live-p` *frame* [Function]
This function returns non-`nil` if the frame *frame* has not been deleted. The possible non-`nil` return values are like those of `framep`. See Chapter 30 [Frames], page 787.

Some window managers provide a command to delete a window. These work by sending a special message to the program that operates the window. When Emacs gets one of these commands, it generates a `delete-frame` event, whose normal definition is a command that calls the function `delete-frame`. See Section 22.7.12 [Misc Events], page 451.

`delete-other-frames` **&optional** *frame iconify* [Command]
This command deletes all frames on *frame*'s terminal, except *frame*. If *frame* uses another frame's minibuffer, that minibuffer frame is left untouched. The argument *frame* must specify a live frame and defaults to the selected frame. Internally, this command works by calling `delete-frame` with *force* `nil` for all frames that shall be deleted.

This function does not delete any of *frame*'s child frames (see Section 30.14 [Child Frames], page 833). If *frame* is a child frame, it deletes *frame*'s siblings only.

With the prefix argument *iconify*, the frames are iconified rather than deleted.

30.8 Finding All Frames

`frame-list` [Function]
This function returns a list of all the live frames, i.e., those that have not been deleted. It is analogous to `buffer-list` for buffers, and includes frames on all terminals. The list that you get is newly created, so modifying the list doesn't have any effect on the internals of Emacs.

`visible-frame-list` [Function]
This function returns a list of just the currently visible frames. See Section 30.11 [Visibility of Frames], page 831. Frames on text terminals always count as visible, even though only the selected one is actually displayed.

`frame-list-z-order` **&optional** *display* [Function]
This function returns a list of Emacs' frames, in Z (stacking) order (see Section 30.12 [Raising and Lowering], page 832). The optional argument *display* specifies which display to poll. *display* should be either a frame or a display name (a string). If omitted or `nil`, that stands for the selected frame's display. It returns `nil` if *display* contains no Emacs frame.

Frames are listed from topmost (first) to bottommost (last). As a special case, if *display* is non-`nil` and specifies a live frame, it returns the child frames of that frame in Z (stacking) order.

This function is not meaningful on text terminals.

`next-frame` **&optional** *frame minibuf* [Function]
This function lets you cycle conveniently through all the frames on a specific terminal from an arbitrary starting point. It returns the frame following *frame*, in the list of all live frames, on *frame*'s terminal. The argument *frame* must specify a live frame and defaults to the selected frame. It never returns a frame whose `no-other-frame` parameter (see Section 30.4.3.6 [Frame Interaction Parameters], page 814) is non-`nil`.

The second argument, *minibuf*, says which frames to consider when deciding what the next frame should be:

`nil` Consider all frames except minibuffer-only frames.

`visible` Consider only visible frames.

0 Consider only visible or iconified frames.

a window Consider only the frames using that particular window as their minibuffer window.

anything else
Consider all frames.

`previous-frame` **&optional** *frame minibuf* [Function]
Like `next-frame`, but cycles through all frames in the opposite direction.

See also `next-window` and `previous-window`, in Section 29.10 [Cyclic Window Ordering], page 719.

Some Lisp programs need to find one or more frames that satisfy a given criteria. The function `filtered-frame-list` is provided for this purpose.

`filtered-frame-list` *predicate* [Function]
This function returns the list of all the live frames which satisfy the specified *predicate*. The argument *predicate* must be a function of one argument, a frame to be tested against the filtering criteria, and should return non-`nil` if the frame satisfies the criteria.

30.9 Minibuffers and Frames

Normally, each frame has its own minibuffer window at the bottom, which is used whenever that frame is selected. You can get that window with the function `minibuffer-window` (see Section 21.11 [Minibuffer Windows], page 419).

However, you can also create a frame without a minibuffer. Such a frame must use the minibuffer window of some other frame. That other frame will serve as *surrogate minibuffer frame* for this frame and cannot be deleted via `delete-frame` (see Section 30.7 [Deleting Frames], page 824) as long as this frame is live.

When you create the frame, you can explicitly specify its minibuffer window (in some other frame) with the `minibuffer` frame parameter (see Section 30.4.3.5 [Buffer Parameters], page 814). If you don't, then the minibuffer is found in the frame which is the value of the variable `default-minibuffer-frame`. Its value should be a frame that does have a minibuffer.

If you use a minibuffer-only frame, you might want that frame to raise when you enter the minibuffer. If so, set the variable `minibuffer-auto-raise` to `t`. See Section 30.12 [Raising and Lowering], page 832.

`default-minibuffer-frame` [Variable]
: This variable specifies the frame to use for the minibuffer window, by default. It does not affect existing frames. It is always local to the current terminal and cannot be buffer-local. See Section 30.2 [Multiple Terminals], page 789.

30.10 Input Focus

At any time, one frame in Emacs is the *selected frame*. The selected window (see Section 29.3 [Selecting Windows], page 698) always resides on the selected frame.

When Emacs displays its frames on several terminals (see Section 30.2 [Multiple Terminals], page 789), each terminal has its own selected frame. But only one of these is *the* selected frame: it's the frame that belongs to the terminal from which the most recent input came. That is, when Emacs runs a command that came from a certain terminal, the selected frame is the one of that terminal. Since Emacs runs only a single command at any given time, it needs to consider only one selected frame at a time; this frame is what we call *the selected frame* in this manual. The display on which the selected frame is shown is the *selected frame's display*.

`selected-frame` [Function]
: This function returns the selected frame.

Some window systems and window managers direct keyboard input to the window object that the mouse is in; others require explicit clicks or commands to *shift the focus* to various window objects. Either way, Emacs automatically keeps track of which frames have focus. To explicitly switch to a different frame from a Lisp function, call `select-frame-set-input-focus`.

The plural "frames" in the previous paragraph is deliberate: while Emacs itself has only one selected frame, Emacs can have frames on many different terminals (recall that a connection to a window system counts as a terminal), and each terminal has its own idea of which frame has input focus. Under the X Window System, where user input is organized into individual "seats" of input, each seat in turn can have its own specific input focus. When you set the input focus to a frame, you set the focus for that frame's terminal on the last seat which interacted with Emacs, but frames on other terminals and seats may still remain focused.

If the input focus is set before any user interaction has occurred on the specified terminal, then the X server picks a random seat (normally the one with the lowest number) and sets the input focus there.

Lisp programs can switch frames temporarily by calling the function `select-frame`. This does not alter the window system's concept of focus; rather, it escapes from the window manager's control until that control is somehow reasserted.

When using a text terminal, only one frame can be displayed at a time on the terminal, so after a call to `select-frame`, the next redisplay actually displays the newly selected frame. This frame remains selected until a subsequent call to `select-frame`. Each frame

on a text terminal has a number which appears in the mode line before the buffer name (see Section 24.4.4 [Mode Line Variables], page 555).

`select-frame-set-input-focus` *frame* **&optional** *norecord* [Function]
: This function selects *frame*, raises it (should it happen to be obscured by other frames) and tries to give it the window system's focus. On a text terminal, the next redisplay displays the new frame on the entire terminal screen. The optional argument *norecord* has the same meaning as for `select-frame` (see below). The return value of this function is not significant.

Ideally, the function described next should focus a frame without also raising it above other frames. Unfortunately, many window-systems or window managers may refuse to comply.

`x-focus-frame` *frame* **&optional** *noactivate* [Function]
: This function gives *frame* the focus of the X server without necessarily raising it. *frame* `nil` means use the selected frame. Under X, the optional argument *noactivate*, if non-`nil`, means to avoid making *frame*'s window-system window the "active" window which should insist a bit more on avoiding to raise *frame* above other frames.

  On MS-Windows the *noactivate* argument has no effect. However, if *frame* is a child frame (see Section 30.14 [Child Frames], page 833), this function usually focuses *frame* without raising it above other child frames.

  If there is no window system support, this function does nothing.

`select-frame` *frame* **&optional** *norecord* [Command]
: This function selects frame *frame*, temporarily disregarding the focus of the X server if any. The selection of *frame* lasts until the next time the user does something to select a different frame, or until the next time this function is called. (If you are using a window system, the previously selected frame may be restored as the selected frame after return to the command loop, because it still may have the window system's input focus.)

  The specified *frame* becomes the selected frame, and its terminal becomes the selected terminal. This function then calls `select-window` as a subroutine, passing the window selected within *frame* as its first argument and *norecord* as its second argument (hence, if *norecord* is non-`nil`, this avoids changing the order of recently selected windows and the buffer list). See Section 29.3 [Selecting Windows], page 698.

  This function returns *frame*, or `nil` if *frame* has been deleted.

  In general, you should never use `select-frame` in a way that could switch to a different terminal without switching back when you're done.

Emacs cooperates with the window system by arranging to select frames as the server and window manager request. When a window system informs Emacs that one of its frames has been selected, Emacs internally generates a *focus-in* event. When an Emacs frame is displayed on a text-terminal emulator, such as `xterm`, which supports reporting of focus-change notification, the focus-in and focus-out events are available even for text-mode frames. Focus events are normally handled by `handle-focus-in`.

`handle-focus-in` *event* [Command]

This function handles focus-in events from window systems and terminals that support explicit focus notifications. It updates the per-frame focus flags that `frame-focus-state` queries and calls `after-focus-change-function`. In addition, it generates a `switch-frame` event in order to switch the Emacs notion of the selected frame to the frame most recently focused in some terminal. It's important to note that this switching of the Emacs selected frame to the most recently focused frame does not mean that other frames do not continue to have the focus in their respective terminals. Do not invoke this function yourself: instead, attach logic to `after-focus-change-function`.

`handle-switch-frame` *frame* [Command]

This function handles a switch-frame event, which Emacs generates for itself upon focus notification or under various other circumstances involving an input event arriving at a different frame from the last event. Do not invoke this function yourself.

`redirect-frame-focus` *frame* **&optional** *focus-frame* [Function]

This function redirects focus from *frame* to *focus-frame*. This means that *focus-frame* will receive subsequent keystrokes and events intended for *frame*. After such an event, the value of `last-event-frame` will be *focus-frame*. Also, switch-frame events specifying *frame* will instead select *focus-frame*.

If *focus-frame* is omitted or `nil`, that cancels any existing redirection for *frame*, which therefore once again receives its own events.

One use of focus redirection is for frames that don't have minibuffers. These frames use minibuffers on other frames. Activating a minibuffer on another frame redirects focus to that frame. This puts the focus on the minibuffer's frame, where it belongs, even though the mouse remains in the frame that activated the minibuffer.

Selecting a frame can also change focus redirections. Selecting frame `bar`, when `foo` had been selected, changes any redirections pointing to `foo` so that they point to `bar` instead. This allows focus redirection to work properly when the user switches from one frame to another using `select-window`.

This means that a frame whose focus is redirected to itself is treated differently from a frame whose focus is not redirected. `select-frame` affects the former but not the latter.

The redirection lasts until `redirect-frame-focus` is called to change it.

`frame-focus-state` *frame* [Function]

This function retrieves the last known focus state of *frame*.

It returns `nil` if the frame is known not to be focused, `t` if the frame is known to be focused, or `unknown` if Emacs does not know the focus state of the frame. (You may see this last state in TTY frames running on terminals that do not support explicit focus notifications.)

`after-focus-change-function` [Variable]

This function is called with no arguments when Emacs notices that a frame may have gotten or lost focus. Focus events are delivered asynchronously, and may not

be delivered in the expected order, so code that wants to do something depending on the state of focused frames have go through all the frames and check.

For instance, here's a simple example function that sets the background color based on whether the frame has focus or not:

```
(add-function :after after-focus-change-function
              #'my-change-background)
(defun my-change-background ()
  (dolist (frame (frame-list))
    (pcase (frame-focus-state frame)
      (`t (set-face-background 'default "black" frame))
      (`nil (set-face-background 'default "#404040" frame)))))
```

Multiple frames may appear to have input focus simultaneously due to focus event delivery differences, the presence of multiple Emacs terminals, and other factors, and code should be robust in the face of this situation.

Depending on window system, focus events may also be delivered repeatedly and with different focus states before settling to the expected values. Code relying on focus notifications should "debounce" any user-visible updates arising from focus changes, perhaps by deferring work until redisplay.

This function may be called in arbitrary contexts, including from inside `read-event`, so take the same care as you might when writing a process filter.

`focus-follows-mouse` [User Option]

This option informs Emacs whether and how the window manager transfers focus when you move the mouse pointer into a frame. It can have three meaningful values:

`nil` The default value `nil` should be used when your window manager follows a "click-to-focus" policy where you have to click the mouse inside of a frame in order for that frame to gain focus.

`t` The value `t` should be used when your window manager has the focus automatically follow the position of the mouse pointer but a frame that gains focus is not raised automatically and may even remain occluded by other window-system windows.

`auto-raise`
The value `auto-raise` should be used when your window manager has the focus automatically follow the position of the mouse pointer and a frame that gains focus is raised automatically.

If this option is non-`nil`, Emacs moves the mouse pointer to the frame selected by `select-frame-set-input-focus`. That function is used by a number of commands like, for example, `other-frame` and `pop-to-buffer`.

The distinction between the values `t` and `auto-raise` is not needed for "normal" frames because the window manager usually takes care of raising them. It is useful to automatically raise child frames via `mouse-autoselect-window` (see Section 29.25 [Mouse Window Auto-selection], page 776).

Note that this option does not distinguish "sloppy" focus (where the frame that previously had focus retains focus as long as the mouse pointer does not move into

another window-system window) from "strict" focus (where a frame immediately loses focus when it's left by the mouse pointer). Neither does it recognize whether your window manager supports delayed focusing or auto-raising where you can explicitly specify the time until a new frame gets focus or is auto-raised.

You can supply a "focus follows mouse" policy for individual Emacs windows by customizing the variable `mouse-autoselect-window` (see Section 29.25 [Mouse Window Auto-selection], page 776).

30.11 Visibility of Frames

A frame on a graphical display may be *visible*, *invisible*, or *iconified*. If it is visible, its contents are displayed in the usual manner. If it is iconified, its contents are not displayed, but there is a little icon somewhere to bring the frame back into view (some window managers refer to this state as *minimized* rather than *iconified*, but from Emacs' point of view they are the same thing). If a frame is invisible, it is not displayed at all.

The concept of visibility is strongly related to that of (un-)mapped frames. A frame (or, more precisely, its window-system window) is and becomes *mapped* when it is displayed for the first time and whenever it changes its state of visibility from `iconified` or `invisible` to `visible`. Conversely, a frame is and becomes *unmapped* whenever it changes its status from `visible` to `iconified` or `invisible`.

Visibility is meaningless on text terminals, since only the selected frame is actually displayed in any case.

`frame-visible-p` *frame* [Function]

This function returns the visibility status of frame *frame*. The value is `t` if *frame* is visible, `nil` if it is invisible, and `icon` if it is iconified.

On a text terminal, all frames are considered visible for the purposes of this function, even though only one frame is displayed. See Section 30.12 [Raising and Lowering], page 832.

`iconify-frame` **&optional** *frame* [Command]

This function iconifies frame *frame*. If you omit *frame*, it iconifies the selected frame. This usually makes all child frames of *frame* (and their descendants) invisible (see Section 30.14 [Child Frames], page 833).

`make-frame-visible` **&optional** *frame* [Command]

This function makes frame *frame* visible. If you omit *frame*, it makes the selected frame visible. This does not raise the frame, but you can do that with `raise-frame` if you wish (see Section 30.12 [Raising and Lowering], page 832).

Making a frame visible usually makes all its child frames (and their descendants) visible as well (see Section 30.14 [Child Frames], page 833).

`make-frame-invisible` **&optional** *frame force* [Command]

This function makes frame *frame* invisible. If you omit *frame*, it makes the selected frame invisible. Usually, this makes all child frames of *frame* (and their descendants) invisible too (see Section 30.14 [Child Frames], page 833).

Unless *force* is non-`nil`, this function refuses to make *frame* invisible if all other frames are invisible.

The visibility status of a frame is also available as a frame parameter. You can read or change it as such. See Section 30.4.3.8 [Management Parameters], page 816. The user can also iconify and deiconify frames with the window manager. This happens below the level at which Emacs can exert any control, but Emacs does provide events that you can use to keep track of such changes. See Section 22.7.12 [Misc Events], page 451.

`x-double-buffered-p` **&optional** *frame* [Function]
: This function returns non-`nil` if *frame* is currently being rendered with double buffering. *frame* defaults to the selected frame.

30.12 Raising, Lowering and Restacking Frames

Most window systems use a desktop metaphor. Part of this metaphor is the idea that system-level windows (representing, e.g., Emacs frames) are stacked in a notional third dimension perpendicular to the screen surface. The order induced by stacking is total and usually referred to as stacking (or Z-) order. Where the areas of two windows overlap, the one higher up in that order will (partially) cover the one underneath.

You can *raise* a frame to the top of that order or *lower* a frame to its bottom by using the functions `raise-frame` and `lower-frame`. You can *restack* a frame directly above or below another frame using the function `frame-restack`.

Note that all functions described below will respect the adherence of frames (and all other window-system windows) to their respective z-group (see Section 30.4.3.2 [Position Parameters], page 807). For example, you usually cannot lower a frame below that of the desktop window and you cannot raise a frame whose `z-group` parameter is `nil` above the window-system's taskbar or tooltip window.

`raise-frame` **&optional** *frame* [Command]
: This function raises frame *frame* (default, the selected frame) above all other frames belonging to the same or a lower z-group as *frame*. If *frame* is invisible or iconified, this makes it visible. If *frame* is a child frame (see Section 30.14 [Child Frames], page 833), this raises *frame* above all other child frames of its parent.

`lower-frame` **&optional** *frame* [Command]
: This function lowers frame *frame* (default, the selected frame) below all other frames belonging to the same or a higher z-group as *frame*. If *frame* is a child frame (see Section 30.14 [Child Frames], page 833), this lowers *frame* below all other child frames of its parent.

`frame-restack` *frame1 frame2* **&optional** *above* [Function]
: This function restacks *frame1* below *frame2*. This implies that if both frames are visible and their display areas overlap, *frame2* will (partially) obscure *frame1*. If the optional third argument *above* is non-`nil`, this function restacks *frame1* above *frame2*. This means that if both frames are visible and their display areas overlap, *frame1* will (partially) obscure *frame2*.

  Technically, this function may be thought of as an atomic action performed in two steps: The first step removes *frame1*'s window-system window from the display. The second step reinserts *frame1*'s window into the display below (above if *above* is true)

that of *frame2*. Hence the position of *frame2* in its display's Z (stacking) order relative to all other frames excluding *frame1* remains unaltered.

Some window managers may refuse to restack windows.

Note that the effect of restacking will only hold as long as neither of the involved frames is iconified or made invisible. You can use the `z-group` (see Section 30.4.3.2 [Position Parameters], page 807) frame parameter to add a frame to a group of frames permanently shown above or below other frames. As long as a frame belongs to one of these groups, restacking it will only affect its relative stacking position within that group. The effect of restacking frames belonging to different z-groups is undefined. You can list frames in their current stacking order with the function `frame-list-z-order` (see Section 30.8 [Finding All Frames], page 825).

`minibuffer-auto-raise` [User Option]

If this is non-`nil`, activation of the minibuffer raises the frame that the minibuffer window is in.

On window systems, you can also enable auto-raising (on frame selection) or auto-lowering (on frame deselection) using frame parameters. See Section 30.4.3.8 [Management Parameters], page 816.

The concept of raising and lowering frames also applies to text terminal frames. On each text terminal, only the top frame is displayed at any one time.

`tty-top-frame` **&optional** *terminal* [Function]

This function returns the top frame on *terminal*. *terminal* should be a terminal object, a frame (meaning that frame's terminal), or `nil` (meaning the selected frame's terminal). If it does not refer to a text terminal, the return value is `nil`.

30.13 Frame Configurations

A *frame configuration* records the current arrangement of frames, all their properties, and the window configuration of each one. (See Section 29.26 [Window Configurations], page 777.)

`current-frame-configuration` [Function]

This function returns a frame configuration list that describes the current arrangement of frames and their contents.

`set-frame-configuration` *configuration* **&optional** *nodelete* [Function]

This function restores the state of frames described in *configuration*. However, this function does not restore deleted frames.

Ordinarily, this function deletes all existing frames not listed in *configuration*. But if *nodelete* is non-`nil`, the unwanted frames are iconified instead.

30.14 Child Frames

Child frames are objects halfway between windows (see Chapter 29 [Windows], page 692) and "normal" frames. Like windows, they are attached to an owning frame. Unlike windows,

they may overlap each other—changing the size or position of one child frame does not change the size or position of any of its sibling child frames.

By design, operations to make or modify child frames are implemented with the help of frame parameters (see Section 30.4 [Frame Parameters], page 804) without any specialized functions or customizable variables. Note that child frames are meaningful on graphical terminals only.

To create a new child frame or to convert a normal frame into a child frame, set that frame's `parent-frame` parameter (see Section 30.4.3.6 [Frame Interaction Parameters], page 814) to that of an already existing frame. The frame specified by that parameter will then be the frame's parent frame as long as the parameter is not changed or reset. Technically, this makes the child frame's window-system window a child window of the parent frame's window-system window.

The `parent-frame` parameter can be changed at any time. Setting it to another frame *reparents* the child frame. Setting it to another child frame makes the frame a *nested* child frame. Setting it to `nil` restores the frame's status as a top-level frame—a frame whose window-system window is a child of its display's root window.[2]

Since child frames can be arbitrarily nested, a frame can be both a child and a parent frame. Also, the relative roles of child and parent frame may be reversed at any time (though it's usually a good idea to keep the size of a child frame sufficiently smaller than that of its parent). An error will be signaled for the attempt to make a frame an ancestor of itself.

Most window-systems clip a child frame at the native edges (see Section 30.3 [Frame Geometry], page 793) of its parent frame—everything outside these edges is usually invisible. A child frame's `left` and `top` parameters specify a position relative to the top-left corner of its parent's native frame. When the parent frame is resized, this position remains conceptually unaltered.

NS builds do not clip child frames at the parent frame's edges, allowing them to be positioned so they do not obscure the parent frame while still being visible themselves.

Usually, moving a parent frame moves along all its child frames and their descendants as well, keeping their relative positions unaltered. Note that the hook `move-frame-functions` (see Section 30.3.3 [Frame Position], page 800) is run for a child frame only when the position of the child frame relative to its parent frame changes.

When a parent frame is resized, its child frames conceptually retain their previous sizes and their positions relative to the left upper corner of the parent. This means that a child frame may become (partially) invisible when its parent frame shrinks. The parameter `keep-ratio` (see Section 30.4.3.6 [Frame Interaction Parameters], page 814) can be used to resize and reposition a child frame proportionally whenever its parent frame is resized. This may avoid obscuring parts of a frame when its parent frame is shrunk.

A visible child frame always appears on top of its parent frame thus obscuring parts of it, except on NS builds where it may be positioned beneath the parent. This is comparable to the window-system window of a top-level frame which also always appears on top of

[2] On Haiku, child frames are only visible when a parent frame is active, owing to a limitation of the Haiku windowing system. Owing to the same limitation, child frames are only guaranteed to appear above their top-level parent; that is to say, the top-most frame in the hierarchy, which does not have a parent frame.

its parent window—the desktop's root window. When a parent frame is iconified or made invisible (see Section 30.11 [Visibility of Frames], page 831), its child frames are made invisible. When a parent frame is deiconified or made visible, its child frames are made visible.

When a parent frame is about to be deleted (see Section 30.7 [Deleting Frames], page 824), its child frames are recursively deleted before it. There is one exception to this rule: When the child frame serves as a surrogate minibuffer frame (see Section 30.9 [Minibuffers and Frames], page 826) for another frame, it is retained until the parent frame has been deleted. If, at this time, no remaining frame uses the child frame as its minibuffer frame, Emacs will try to delete the child frame too. If that deletion fails for whatever reason, the child frame is made a top-level frame.

Whether a child frame can have a menu or tool bar is window-system or window manager dependent. Most window-systems explicitly disallow menu bars for child frames. It seems advisable to disable both, menu and tool bars, via the frame's initial parameters settings.

Usually, child frames do not exhibit window manager decorations like a title bar or external borders (see Section 30.3 [Frame Geometry], page 793). When the child frame does not show a menu or tool bar, any other of the frame's borders (see Section 30.4.3.4 [Layout Parameters], page 812) can be used instead of the external borders.

In particular, under X (but not when building with GTK+), the frame's outer border can be used. On MS-Windows, specifying a non-zero outer border width will show a one-pixel wide external border. Under all window-systems, the internal border can be used. In either case, it's advisable to disable a child frame's window manager decorations with the `undecorated` frame parameter (see Section 30.4.3.8 [Management Parameters], page 816).

To resize or move an undecorated child frame with the mouse, special frame parameters (see Section 30.4.3.7 [Mouse Dragging Parameters], page 816) have to be used. The internal border of a child frame, if present, can be used to resize the frame with the mouse, provided that frame has a non-`nil` `drag-internal-border` parameter. If set, the `snap-width` parameter indicates the number of pixels where the frame *snaps* at the respective edge or corner of its parent frame.

There are two ways to drag an entire child frame with the mouse: The `drag-with-mode-line` parameter, if non-`nil`, allows to drag a frame without minibuffer window (see Section 21.11 [Minibuffer Windows], page 419) via the mode line area of its bottommost window. The `drag-with-header-line` parameter, if non-`nil`, allows to drag the frame via the header line area of its topmost window.

In order to give a child frame a draggable header or mode line, the window parameters `mode-line-format` and `header-line-format` are handy (see Section 29.27 [Window Parameters], page 779). These allow to remove an unwanted mode line (when `drag-with-header-line` is chosen) and to remove mouse-sensitive areas which might interfere with frame dragging.

When the user drags a frame with a mouse and overshoots, it's easy to drag a frame out of the screen area of its parent. Retrieving such a frame can be hairy once the mouse button has been released. To prevent such a situation, it is advisable to set the frame's `top-visible` or `bottom-visible` parameter (see Section 30.4.3.7 [Mouse Dragging Parameters], page 816).

Set the `top-visible` parameter of a child frame to a number when you intend to allow the user dragging that frame by its header line. Setting `top-visible` to a number

inhibits dragging the top edge of the child frame above the top edge of its parent. Set the `bottom-visible` parameter to a number when you intend to drag that frame via its mode line; this inhibits dragging the bottom edge of the child frame beneath the bottom edge of its parent. In either case, that number also specifies width and height (in pixels) of the area of the child frame that remains visible during dragging.

When a child frame is used for displaying a buffer via `display-buffer-in-child-frame` (see Section 29.13.2 [Buffer Display Action Functions], page 728), the frame's `auto-hide-function` parameter (see Section 30.4.3.6 [Frame Interaction Parameters], page 814) can be set to a function, in order to appropriately deal with the frame when the window displaying the buffer shall be quit.

When a child frame is used during minibuffer interaction, for example, to display completions in a separate window, the `minibuffer-exit` parameter (see Section 30.4.3.6 [Frame Interaction Parameters], page 814) is useful in order to deal with the frame when the minibuffer is exited.

The behavior of child frames deviates from that of top-level frames in a number of other ways as well. Here we sketch a few of them:

- The semantics of maximizing and iconifying child frames is highly window-system dependent. As a rule, applications should never invoke these operations on child frames. By default, invoking `iconify-frame` on a child frame will try to iconify the top-level frame corresponding to that child frame instead. To obtain a different behavior, users may customize the option `iconify-child-frame` described below.
- Raising, lowering and restacking child frames (see Section 30.12 [Raising and Lowering], page 832) or changing the `z-group` (see Section 30.4.3.2 [Position Parameters], page 807) of a child frame changes only the stacking order of child frames with the same parent.
- Many window-systems are not able to change the opacity (see Section 30.4.3.10 [Font and Color Parameters], page 820) of child frames.
- Transferring focus from a child frame to an ancestor that is not its parent by clicking with the mouse in a visible part of that ancestor's window may fail with some window-systems. You may have to click into the direct parent's window-system window first.
- Window managers might not bother to extend their focus follows mouse policy to child frames. Customizing `mouse-autoselect-window` can help in this regard (see Section 29.25 [Mouse Window Auto-selection], page 776).
- Dropping (see Section 30.22 [Drag and Drop], page 844) on child frames is not guaranteed to work on all window-systems. Some will drop the object on the parent frame or on some ancestor instead.

The following two functions can be useful when working with child and parent frames:

`frame-parent` **&optional** *frame* [Function]

This function returns the parent frame of *frame*. The parent frame of *frame* is the Emacs frame whose window-system window is the parent window of *frame*'s window-system window. If such a frame exists, *frame* is considered a child frame of that frame.

This function returns `nil` if *frame* has no parent frame.

`frame-ancestor-p` *ancestor descendant* [Function]
: This functions returns non-`nil` if *ancestor* is an ancestor of *descendant*. *ancestor* is an ancestor of *descendant* when it is either *descendant*'s parent frame or it is an ancestor of *descendant*'s parent frame. Both, *ancestor* and *descendant* must specify live frames.

Note also the function `window-largest-empty-rectangle` (see Section 29.24 [Coordinates and Windows], page 772) which can be used to inscribe a child frame in the largest empty area of an existing window. This can be useful to avoid that a child frame obscures any text shown in that window.

Customizing the following option can be useful to tweak the behavior of `iconify-frame` for child frames.

`iconify-child-frame` [User Option]
: This option tells Emacs how to proceed when it is asked to iconify a child frame. If it is `nil`, `iconify-frame` will do nothing when invoked on a child frame. If it is `iconify-top-level`, Emacs will try to iconify the top-level frame that is the ancestor of this child frame instead. If it is `make-invisible`, Emacs will try to make this child frame invisible instead of iconifying it.

  Any other value means to try iconifying the child frame. Since such an attempt may not be honored by all window managers and can even lead to making the child frame unresponsive to user actions, the default is to iconify the top level frame instead.

30.15 Mouse Tracking

Sometimes it is useful to *track* the mouse, which means to display something to indicate where the mouse is and move the indicator as the mouse moves. For efficient mouse tracking, you need a way to wait until the mouse actually moves.

The convenient way to track the mouse is to ask for events to represent mouse motion. Then you can wait for motion by waiting for an event. In addition, you can easily handle any other sorts of events that may occur. That is useful, because normally you don't want to track the mouse forever—only until some other event, such as the release of a button.

`track-mouse` *body...* [Macro]
: This macro executes *body*, with generation of mouse motion events enabled. Typically, *body* would use `read-event` to read the motion events and modify the display accordingly. See Section 22.7.8 [Motion Events], page 448, for the format of mouse motion events.

  The value of `track-mouse` is that of the last form in *body*. You should design *body* to return when it sees the up-event that indicates the release of the button, or whatever kind of event means it is time to stop tracking. Its value also controls how mouse events are reported while a mouse button is held down: if it is `dropping` or `drag-source`, the motion events are reported relative to the frame underneath the pointer. If there is no such frame, the events will be reported relative to the frame the mouse buttons were first pressed on. In addition, the `posn-window` of the mouse position list will be `nil` if the value is `drag-source`. This is useful to determine if a frame is not directly visible underneath the mouse pointer.

The `track-mouse` macro causes Emacs to generate mouse motion events by binding the variable `track-mouse` to a non-`nil` value. If that variable has the special value `dragging`, it additionally instructs the display engine to refrain from changing the shape of the mouse pointer. This is desirable in Lisp programs that require mouse dragging across large portions of Emacs display, which might otherwise cause the mouse pointer to change its shape according to the display portion it hovers on (see Section 30.19 [Pointer Shape], page 841). Therefore, Lisp programs that need the mouse pointer to retain its original shape during dragging should bind `track-mouse` to the value `dragging` at the beginning of their *body*.

The usual purpose of tracking mouse motion is to indicate on the screen the consequences of pushing or releasing a button at the current position.

In many cases, you can avoid the need to track the mouse by using the `mouse-face` text property (see Section 33.19.4 [Special Properties], page 927). That works at a much lower level and runs more smoothly than Lisp-level mouse tracking.

30.16 Mouse Position

The functions `mouse-position` and `set-mouse-position` give access to the current position of the mouse.

`mouse-position` [Function]
: This function returns a description of the position of the mouse. The value looks like `(`*`frame x . y`*`)`, where *x* and *y* are integers giving the (possibly rounded) position in multiples of the default character size of *frame* (see Section 30.3.2 [Frame Font], page 799) relative to the native position of *frame* (see Section 30.3 [Frame Geometry], page 793).

`mouse-position-function` [Variable]
: If non-`nil`, the value of this variable is a function for `mouse-position` to call. `mouse-position` calls this function just before returning, with its normal return value as the sole argument, and it returns whatever this function returns to it.

  This abnormal hook exists for the benefit of packages like `xt-mouse.el` that need to do mouse handling at the Lisp level.

`tty-menu-calls-mouse-position-function` [Variable]
: If non-`nil`, TTY menus will call `mouse-position-function` as described above. This exists for cases where `mouse-position-function` is not safe to be called by the TTY menus, such as if it could trigger redisplay.

`set-mouse-position` *frame x y* [Function]
: This function *warps the mouse* to position *x*, *y* in frame *frame*. The arguments *x* and *y* are integers, giving the position in multiples of the default character size of *frame* (see Section 30.3.2 [Frame Font], page 799) relative to the native position of *frame* (see Section 30.3 [Frame Geometry], page 793).

  The resulting mouse position is constrained to the native frame of *frame*. If *frame* is not visible, this function does nothing. The return value is not significant.

`mouse-pixel-position` [Function]
: This function is like `mouse-position` except that it returns coordinates in units of pixels rather than units of characters.

`set-mouse-pixel-position` *frame x y* [Function]
: This function warps the mouse like `set-mouse-position` except that *x* and *y* are in units of pixels rather than units of characters.

  The resulting mouse position is not constrained to the native frame of *frame*. If *frame* is not visible, this function does nothing. The return value is not significant.

On a graphical terminal the following two functions allow the absolute position of the mouse cursor to be retrieved and set.

`mouse-absolute-pixel-position` [Function]
: This function returns a cons cell (*x* . *y*) of the coordinates of the mouse cursor position in pixels, relative to a position (0, 0) of the selected frame's display.

`set-mouse-absolute-pixel-position` *x y* [Function]
: This function moves the mouse cursor to the position (*x*, *y*). The coordinates *x* and *y* are interpreted in pixels relative to a position (0, 0) of the selected frame's display.

The following function can tell whether the mouse cursor is currently visible on a frame:

`frame-pointer-visible-p` **&optional** *frame* [Function]
: This predicate function returns non-`nil` if the mouse pointer displayed on *frame* is visible; otherwise it returns `nil`. *frame* omitted or `nil` means the selected frame. This is useful when `make-pointer-invisible` is set to `t`: it allows you to know if the pointer has been hidden. See Section "Mouse Avoidance" in *The Emacs Manual*.

30.17 Pop-Up Menus

A Lisp program can pop up a menu so that the user can choose an alternative with the mouse. On a text terminal, if the mouse is not available, the user can choose an alternative using the keyboard motion keys—`C-n`, `C-p`, or up- and down-arrow keys.

`x-popup-menu` *position menu* [Function]
: This function displays a pop-up menu and returns an indication of what selection the user makes.

  The argument *position* specifies where on the screen to put the top left corner of the menu. It can be either a mouse button event (which says to put the menu where the user actuated the button) or a list of this form:

  ```
  ((xoffset yoffset) window)
  ```

  where *xoffset* and *yoffset* are coordinates, measured in pixels, counting from the top left corner of *window*. *window* may be a window or a frame.

  If *position* is `t`, it means to use the current mouse position (or the top-left corner of the frame if the mouse is not available on a text terminal). If *position* is `nil`, it means to precompute the key binding equivalents for the keymaps specified in *menu*, without actually displaying or popping up the menu.

The argument *menu* says what to display in the menu. It can be a keymap or a list of keymaps (see Section 23.18 [Menu Keymaps], page 511). In this case, the return value is the list of events corresponding to the user's choice. This list has more than one element if the choice occurred in a submenu. (Note that `x-popup-menu` does not actually execute the command bound to that sequence of events.) On text terminals and toolkits that support menu titles, the title is taken from the prompt string of *menu* if *menu* is a keymap, or from the prompt string of the first keymap in *menu* if it is a list of keymaps (see Section 23.18.1 [Defining Menus], page 511).

Alternatively, *menu* can have the following form:

```
(title pane1 pane2...)
```

where each pane is a list of form

```
(title item1 item2...)
```

Each *item* should be a cons cell, `(line . value)`, where *line* is a string and *value* is the value to return if that *line* is chosen. Unlike in a menu keymap, a `nil` *value* does not make the menu item non-selectable. Alternatively, each *item* can be a string rather than a cons cell; this makes a non-selectable menu item.

If the user gets rid of the menu without making a valid choice, for instance by clicking the mouse away from a valid choice or by typing `C-g`, then this normally results in a quit and `x-popup-menu` does not return. But if *position* is a mouse button event (indicating that the user invoked the menu with the mouse) then no quit occurs and `x-popup-menu` returns `nil`.

Usage note: Don't use `x-popup-menu` to display a menu if you could do the job with a prefix key defined with a menu keymap. If you use a menu keymap to implement a menu, `C-h c` and `C-h a` can see the individual items in that menu and provide help for them. If instead you implement the menu by defining a command that calls `x-popup-menu`, the help facilities cannot know what happens inside that command, so they cannot give any help for the menu's items.

The menu bar mechanism, which lets you switch between submenus by moving the mouse, cannot look within the definition of a command to see that it calls `x-popup-menu`. Therefore, if you try to implement a submenu using `x-popup-menu`, it cannot work with the menu bar in an integrated fashion. This is why all menu bar submenus are implemented with menu keymaps within the parent menu, and never with `x-popup-menu`. See Section 23.18.5 [Menu Bar], page 517.

If you want a menu bar submenu to have contents that vary, you should still use a menu keymap to implement it. To make the contents vary, add a hook function to `menu-bar-update-hook` to update the contents of the menu keymap as necessary.

`x-pre-popup-menu-hook` [Variable]

A normal hook run immediately before a pop-up menu is displayed, either directly by calling `x-popup-menu`, or through a menu keymap. It won't be called if `x-popup-menu` returns for some other reason without displaying a pop-up menu.

30.18 Dialog Boxes

A dialog box is a variant of a pop-up menu—it looks a little different, it always appears in the center of a frame, and it has just one level and one or more buttons. The main use of dialog boxes is for asking questions that the user can answer with "yes", "no", and a few other alternatives. With a single button, they can also force the user to acknowledge important information. The functions `y-or-n-p` and `yes-or-no-p` use dialog boxes instead of the keyboard, when called from commands invoked by mouse clicks.

`x-popup-dialog` *position contents* **&optional** *header* [Function]

This function displays a pop-up dialog box and returns an indication of what selection the user makes. The argument *contents* specifies the alternatives to offer; it has this format:

```
(title (string . value)...)
```

which looks like the list that specifies a single pane for `x-popup-menu`.

The return value is *value* from the chosen alternative.

As for `x-popup-menu`, an element of the list may be just a string instead of a cons cell `(string . value)`. That makes a box that cannot be selected.

If `nil` appears in the list, it separates the left-hand items from the right-hand items; items that precede the `nil` appear on the left, and items that follow the `nil` appear on the right. If you don't include a `nil` in the list, then approximately half the items appear on each side.

Dialog boxes always appear in the center of a frame; the argument *position* specifies which frame. The possible values are as in `x-popup-menu`, but the precise coordinates or the individual window don't matter; only the frame matters.

If *header* is non-`nil`, the frame title for the box is '`Information`', otherwise it is '`Question`'. The former is used for `message-box` (see [message-box], page 1138). (On text terminals, the box title is not displayed.)

In some configurations, Emacs cannot display a real dialog box; so instead it displays the same items in a pop-up menu in the center of the frame.

If the user gets rid of the dialog box without making a valid choice, for instance using the window manager, then this produces a quit and `x-popup-dialog` does not return.

30.19 Pointer Shape

You can specify the mouse pointer style for particular text or images using the `pointer` text property, and for images with the `:pointer` and `:map` image properties. The values you can use in these properties are in the table below. The actual shapes may vary between systems; the descriptions are examples.

`text`
`nil` The usual mouse pointer style used over text (an "I"-like shape).

`arrow`
`vdrag`
`modeline` An arrow that points north-west.

`hand` A hand that points upwards.

`hdrag` A right-left arrow.

`nhdrag` An up-down arrow.

`hourglass`
A rotating ring.

Over void parts of the window (parts that do not correspond to any of the buffer contents), the mouse pointer usually uses the `arrow` style, but you can specify a different style (one of those above) by setting `void-text-area-pointer`.

`void-text-area-pointer` [User Option]
This variable specifies the mouse pointer style for void text areas. These include the areas after the end of a line or below the last line in the buffer. The default is to use the `arrow` (non-text) pointer style.

When using some window systems, you can specify what the `text` pointer style really looks like by setting the variable `x-pointer-shape`.

`x-pointer-shape` [Variable]
This variable specifies the pointer shape to use ordinarily in the Emacs frame, for the `text` pointer style.

`x-sensitive-text-pointer-shape` [Variable]
This variable specifies the pointer shape to use when the mouse is over mouse-sensitive text.

These variables affect newly created frames. They do not normally affect existing frames; however, if you set the mouse color of a frame, that also installs the current value of those two variables. See Section 30.4.3.10 [Font and Color Parameters], page 820.

The values you can use, to specify either of these pointer shapes, are defined in the file `lisp/term/x-win.el`. Use *M-x apropos* `RET` *x-pointer* `RET` to see a list of them.

30.20 Window System Selections

In window systems, such as X, data can be transferred between different applications by means of *selections*. X defines an arbitrary number of *selection types*, each of which can store its own data; however, only three are commonly used: the *clipboard*, *primary selection*, and *secondary selection*. Other window systems support only the clipboard. See Section "Cut and Paste" in *The GNU Emacs Manual*, for Emacs commands that make use of these selections. This section documents the low-level functions for reading and setting window-system selections.

`gui-set-selection` *type data* [Command]
This function sets a window-system selection. It takes two arguments: a selection type *type*, and the value to assign to it, *data*.

type should be a symbol; it is usually one of `PRIMARY`, `SECONDARY` or `CLIPBOARD`. These are symbols with upper-case names, in accord with X Window System conventions. If *type* is `nil`, that stands for `PRIMARY`.

If *data* is `nil`, it means to clear out the selection. Otherwise, *data* may be a string, a symbol, an integer, an overlay, or a cons of two markers pointing to the same buffer.

An overlay or a pair of markers stands for text in the overlay or between the markers. The argument *data* may also be a vector of valid non-vector selection values.

If *data* is a string, then its text properties can specify values used for individual data types. For example, if *data* has a property named `text/uri-list`, then a call to `gui-get-selection` with the data type `text/uri-list` will result in the value of that property being used instead of *data* itself.

This function returns *data*.

`gui-get-selection` **&optional** *type data-type* [Function]
This function accesses selections set up by Emacs or by other programs. It takes two optional arguments, *type* and *data-type*. The default for *type*, the selection type, is `PRIMARY`.

The *data-type* argument specifies the form of data conversion to use, to convert the raw data obtained from another program into Lisp data. Meaningful values include `TEXT`, `STRING`, `UTF8_STRING`, `TARGETS`, `LENGTH`, `DELETE`, `FILE_NAME`, `CHARACTER_POSITION`, `NAME`, `LINE_NUMBER`, `COLUMN_NUMBER`, `OWNER_OS`, `HOST_NAME`, `USER`, `CLASS`, `ATOM`, and `INTEGER`. (These are symbols with upper-case names in accord with X conventions.) The default for *data-type* is `STRING`. Window systems other than X usually support only a small subset of these types, in addition to `STRING`.

`selection-coding-system` [User Option]
This variable specifies the coding system to use when reading and writing selections or the clipboard. See Section 34.10 [Coding Systems], page 980. The default is `compound-text-with-extensions`, which converts to the text representation that X11 normally uses.

When Emacs runs on MS-Windows, it does not implement X selections in general, but it does support the clipboard. `gui-get-selection` and `gui-set-selection` on MS-Windows support the text data type only; if the clipboard holds other types of data, Emacs treats the clipboard as empty. The supported data type is `STRING`.

For backward compatibility, there are obsolete aliases `x-get-selection` and `x-set-selection`, which were the names of `gui-get-selection` and `gui-set-selection` before Emacs 25.1.

30.21 Yanking Media

If you choose, for instance, "Copy Image" in a web browser, that image is put onto the clipboard, and Emacs can access it via `gui-get-selection`. But in general, inserting image data into an arbitrary buffer isn't very useful—you can't really do much with it by default.

So Emacs has a system to let modes register handlers for these "complicated" selections.

`yank-media-handler` *types handler* [Function]
types can be a MIME media type symbol, a regexp to match these, or a list of these symbols and regexps. For instance:

```
(yank-media-handler 'text/html #'my-html-handler)
(yank-media-handler "image/.*" #'my-image-handler)
```

A mode can register as many handlers as required.

The *handler* function is called with two parameters: The MIME media type symbol and the data (as a string). The handler should then insert the object into the buffer, or save it, or do whatever is appropriate for the mode.

The `yank-media` command will consult the registered handlers in the current buffer, compare that with the available media types on the clipboard, and then pass on the matching selection to the handler (if any). If there's more than one matching selection, the user is queried first.

The `yank-media-types` command can be used to explore the clipboard/primary selection. It lists all the media types that are currently available, and can be handy when creating handlers—to see what data is actually available. Some applications put a surprising amount of different data types on the clipboard.

30.22 Drag and Drop

When the user drops something from another application over Emacs, Emacs will try to insert any text and open any URL that was dropped. If text was dropped, then it will always be inserted at the location of the mouse pointer where the drop happened, or saved in the kill ring if insertion failed, which could happen if the buffer was read-only. If a URL was dropped instead, then Emacs will first try to call an appropriate handler function by matching the URL against regexps defined in the variable `dnd-protocol-alist`, and then against those defined in the variables `browse-url-handlers` and `browse-url-default-handlers`. Should no suitable handler be located, Emacs will fall back to inserting the URL as plain text.

`dnd-protocol-alist` [Variable]

This variable is a list of cons cells of the form `(`*`pattern`*` . `*`action`*`)`. *pattern* is a regexp that URLs are matched against after being dropped. *action* is a function that is called with two arguments, should a URL being dropped match *pattern*: the URL being dropped, and the action being performed for the drop, which is one of the symbols `copy`, `move`, `link`, `private` or `ask`.

If *action* is *private*, then it means the program that initiated the drop wants Emacs to perform an unspecified action with the URL; a reasonable action to perform in that case is to open the URL or copy its contents into the current buffer. Otherwise, *action* has the same meaning as the *action* argument to `dnd-begin-file-drag`.

Emacs implements receiving text and URLs individually for each window system, and does not by default support receiving other kinds of data as drops. To support receiving other kinds of data, use the X-specific interface described below.

When a user drags something from another application over Emacs under the X Window System, that other application expects Emacs to tell it if Emacs understands the data being dragged. The function in the variable `x-dnd-test-function` is called by Emacs to determine what to reply to any such inquiry. The default value is `x-dnd-default-test-function`, which accepts drops if the type of the data to be dropped is present in `x-dnd-known-types`. Changing the variables `x-dnd-test-function` and `x-dnd-known-types` can make Emacs accept or reject drops based on some other criteria.

If you want to change the way Emacs receives drops of different data types, or you want to enable it to understand a new type, change the variable `x-dnd-types-alist`. Doing so correctly requires detailed knowledge of what data types other applications use for drag and drop.

These data types are typically implemented as special data types that can be obtained from an X selection provided by the other application. In most cases, they are either the same data types that are typically accepted by `gui-set-selection`, or MIME types, depending on the specific drag-and-drop protocol being used. For example, the data type used for plain text may be either `"STRING"` or `"text/plain"`.

When Emacs runs on X window system, it supports the X Direct Save (XDS) protocol, which allows users to save a file by dragging and dropping it onto an Emacs window, such as a Dired window. To comply with the unique requirements of XDS, these drag-and-drop requests are processed specially: instead of being handled according to `x-dnd-types-alist`, they are handled by the *direct-save function* that is the value of the variable `x-dnd-direct-save-function`. The value should be a function of two arguments, *need-name* and *filename*. The XDS protocol uses a two-step procedure for dragging files:

1. The application from which the file is dragged asks Emacs to provide the full file name under which to save the file. For this purpose, the direct-save function is called with its first argument *need-name* non-`nil`, and the second argument *filename* set to the basename of the file to be saved. It should return the fully-expanded absolute file name under which to save the file. For example, if a file is dragged to a Dired window, the natural directory for the file is the directory of the file shown at location of the drop. If saving the file is not possible for some reason, the function should return `nil`, which will cancel the drag-and-drop operation.
2. The application from which the file is dragged saves the file under the name returned by the first call to the direct-save function. If it succeeds in saving the file, the direct-save function is called again, this time with the first argument *need-name* set to `nil` and the second argument *filename* set to the full absolute name of the saved file. The function is then expected to do whatever is needed given the fact that file was saved. For example, Dired should update the directory on display by showing the new file there.

The default value of `x-dnd-direct-save-function` is `x-dnd-save-direct`.

`x-dnd-save-direct` *need-name filename* [Function]

When called with the *need-name* argument non-`nil`, this function prompts the user for the absolute file name under which it should be saved. If the specified file already exists, it additionally asks the user whether to overwrite it, and returns the absolute file name only if the user confirms the overwriting.

When called with the *need-name* argument `nil`, it reverts the Dired listing if the current buffer is in Dired mode or one of its descendants, and otherwise visits the file by calling `find-file` (see Section 26.1.1 [Visiting Functions], page 610).

`x-dnd-save-direct-immediately` *need-name filename* [Function]

This function works like `x-dnd-save-direct`, but when called with its *need-name* argument non-`nil`, it doesn't prompt the user for the full name of the file to be saved; instead, it returns its argument *filename* expanded against the current buffer's

default directory (see Section 26.9.4 [File Name Expansion], page 641). (It still asks for confirmation if a file by that name already exists in the default directory.)

On capable window systems, Emacs also supports dragging contents from its frames to windows of other applications.

`dnd-begin-text-drag` *text* **&optional** *frame action allow-same-frame* [Function]
This function begins dragging text from *frame* to another program (known as the *drop target*), and returns the result of drag-and-drop operation when the text is dropped or the drag-and-drop operation is canceled. *text* is the text that will be inserted by the drop target.

action must be one of the symbols `copy` or `move`, where `copy` means that *text* should be inserted by the drop target, and `move` means the same as `copy`, but in addition the caller may have to delete *text* from its source as explained below.

frame is the frame where the mouse is currently held down, or `nil`, which means to use the selected frame. This function may return immediately if no mouse buttons are held down, so it should be only called immediately after a `down-mouse-1` or similar event (see Section 22.7.3 [Mouse Events], page 443), with *frame* set to the frame where that event was generated (see Section 22.7.4 [Click Events], page 443).

allow-same-frame specifies whether or not drops on top of *frame* itself are to be ignored.

The return value specifies the action that the drop target actually performed, and optionally what the caller should do. It can be one of the following symbols:

`copy` The drop target inserted the dropped text.

`move` The drop target inserted the dropped text, but in addition the caller should delete *text* from wherever it originated, such as its buffer.

`private` The drop target performed some other unspecified action.

`nil` The drag-and-drop operation was canceled.

`dnd-begin-file-drag` *file* **&optional** *frame action allow-same-frame* [Function]
This function begins dragging *file* from *frame* to another program, and returns the result of the drag-and-drop operation when the file is dropped or the drag-and-drop operation is canceled.

If *file* is a remote file, then a temporary copy will be made.

action must be one of the symbols `copy`, `move` or `link`, where `copy` means that *file* should be opened or copied by the drop target, `move` means the drop target should move the file to another location, and `link` means the drop target should create a symbolic link to *file*. It is an error to specify `link` as the action if *file* is a remote file.

frame and *allow-same-frame* have the same meaning as in `dnd-begin-text-drag`.

The return value is the action that the drop target actually performed, which can be one of the following symbols:

`copy` The drop target opened or copied *file* to a different location.

`move` The drop target moved *file* to a different location.

`link` The drop target (usually a file manager) created a symbolic link to *file*.

`private` The drop target performed some other unspecified action.

`nil` The drag-and-drop operation was canceled.

`dnd-begin-drag-files` *files* **&optional** *frame action allow-same-frame* [Function]

This function is like `dnd-begin-file-drag`, except that *files* is a list of files. If the drop target doesn't support dropping multiple files, then the first file will be used instead.

`dnd-direct-save` *file name* **&optional** *frame allow-same-frame* [Function]

This function is similar to `dnd-begin-file-drag` (with the default action of copy), but instead of specifying the action you specify the name of the copy created by the target program in `name`.

The high-level interfaces described above are implemented on top of a lower-level primitive. If you need to drag content other than files or text, use the low-level interface `x-begin-drag` instead. However, using it will require detailed knowledge of the data types and actions used by the programs to transfer content via drag-and-drop on each platform you want to support.

`x-begin-drag` *targets* **&optional** *action frame return-frame allow-current-frame follow-tooltip* [Function]

This function begins a drag from *frame*, and returns when the drag-and-drop operation ends, either because the drop was successful, or because the drop was rejected. The drop occurs when all mouse buttons are released on top of an X window other than *frame* (the *drop target*), or any X window if *allow-current-frame* is non-`nil`. If no mouse buttons are held down when the drag-and-drop operation begins, this function may immediately return `nil`.

targets is a list of strings describing selection targets, much like the *data-type* argument to `gui-get-selection`, that the drop target can request from Emacs (see Section 30.20 [Window System Selections], page 842).

action is a symbol describing the action recommended to the target. It can either be `XdndActionCopy`, which means to copy the contents of the selection `XdndSelection` to the drop target; or `XdndActionMove`, which means copy as with `XdndActionCopy`, and in addition the caller should delete whatever was stored in that selection after copying it.

action may also be an alist which associates between symbols describing the available actions, and strings that the drop target is expected to present to the user to choose between the available actions.

If *return-frame* is non-`nil` and the mouse moves over an Emacs frame after first moving out of *frame*, then the frame to which the mouse moves will be returned immediately. If *return-frame* is the symbol `now`, then any frame underneath the mouse pointer will be returned without waiting for the mouse to first move out of *frame*. *return-frame* is useful when you want to treat dragging content from one frame to another specially, while also being able to drag content to other programs, but it is not guaranteed to work on all systems and with all window managers.

If *follow-tooltip* is non-`nil`, the position of any tooltip (such as one shown by `tooltip-show`) will follow the location of the mouse pointer whenever it moves during the drag-and-drop operation. The tooltip will be hidden once all mouse buttons are released.

If the drop was rejected or no drop target was found, this function returns `nil`. Otherwise, it returns a symbol describing the action the target chose to perform, which can differ from *action* if that isn't supported by the drop target. `XdndActionPrivate` is also a valid return value in addition to `XdndActionCopy` and `XdndActionMove`; it means that the drop target chose to perform an unspecified action, and no further processing is required by the caller.

The caller must cooperate with the target to fully perform the action chosen by the target. For example, callers should delete the buffer text that was dragged if this function returns `XdndActionMove`.

On X Windows, several different drag-and-drop protocols are supported by `x-begin-drag`. When dragging content that is known to not be supported by a specific drag-and-drop protocol, it might be desirable to turn that protocol off, by changing the values of the following variables:

`x-dnd-disable-motif-protocol` [Variable]

When this is non-`nil`, the Motif drag and drop protocols are disabled, and dropping onto programs that only understand them will not work.

`x-dnd-use-offix-drop` [Variable]

When this is `nil`, the OffiX (old KDE) drag and drop protocol is disabled. When this is the symbol `files`, the OffiX protocol will only be used if `"FILE_NAME"` is one of the targets given to `x-begin-drag`. Any other value means to use the OffiX protocol to drop all supported content.

`x-dnd-use-unsupported-drop` [Variable]

When one of the `"STRING"`, `"UTF8_STRING"`, `"COMPOUND_TEXT"` or `"TEXT"` targets is present in the list given to `x-begin-drag`, Emacs will try to use synthesized mouse events and the primary selection to insert the text if the drop target doesn't support any drag-and-drop protocol at all.

A side effect is that Emacs will become the owner of the primary selection upon such a drop. If that is not desired, then the drop emulation can be disabled by setting this variable to `nil`.

30.23 Color Names

A color name is text (usually in a string) that specifies a color. Symbolic names such as '`black`', '`white`', '`red`', etc., are allowed; use *`M-x list-colors-display`* to see a list of defined names. You can also specify colors numerically in forms such as '`#rgb`' and '`RGB:r/g/b`', where *r* specifies the red level, *g* specifies the green level, and *b* specifies the blue level. You can use either one, two, three, or four hex digits for *r*; then you must use the same number of hex digits for all *g* and *b* as well, making either 3, 6, 9 or 12 hex digits in all. (See the documentation of the X Window System for more details about numerical RGB specification of colors.)

These functions provide a way to determine which color names are valid, and what they look like. In some cases, the value depends on the *selected frame*, as described below; see Section 30.10 [Input Focus], page 827, for the meaning of the term "selected frame".

To read user input of color names with completion, use `read-color` (see Section 21.6.4 [High-Level Completion], page 403).

`color-defined-p` *color* **&optional** *frame* [Function]

This function reports whether a color name is meaningful. It returns `t` if so; otherwise, `nil`. The argument *frame* says which frame's display to ask about; if *frame* is omitted or `nil`, the selected frame is used.

Note that this does not tell you whether the display you are using really supports that color. When using X, you can ask for any defined color on any kind of display, and you will get some result—typically, the closest it can do. To determine whether a frame can really display a certain color, use `color-supported-p` (see below).

This function used to be called `x-color-defined-p`, and that name is still supported as an alias.

`defined-colors` **&optional** *frame* [Function]

This function returns a list of the color names that are defined and supported on frame *frame* (default, the selected frame). If *frame* does not support colors, the value is `nil`.

This function used to be called `x-defined-colors`, and that name is still supported as an alias.

`color-supported-p` *color* **&optional** *frame background-p* [Function]

This returns `t` if *frame* can really display the color *color* (or at least something close to it). If *frame* is omitted or `nil`, the question applies to the selected frame.

Some terminals support a different set of colors for foreground and background. If *background-p* is non-`nil`, that means you are asking whether *color* can be used as a background; otherwise you are asking whether it can be used as a foreground.

The argument *color* must be a valid color name.

`color-gray-p` *color* **&optional** *frame* [Function]

This returns `t` if *color* is a shade of gray, as defined on *frame*'s display. If *frame* is omitted or `nil`, the question applies to the selected frame. If *color* is not a valid color name, this function returns `nil`.

`color-values` *color* **&optional** *frame* [Function]

This function returns a value that describes what *color* should ideally look like on *frame*. If *color* is defined, the value is a list of three integers, which give the amount of red, the amount of green, and the amount of blue. Each integer ranges in principle from 0 to 65535, but some displays may not use the full range. This three-element list is called the *rgb values* of the color.

If *color* is not defined, the value is `nil`.

```
(color-values "black")
     ⇒ (0 0 0)
(color-values "white")
```

```
     ⇒ (65535 65535 65535)
(color-values "red")
     ⇒ (65535 0 0)
(color-values "pink")
     ⇒ (65535 49344 52171)
(color-values "hungry")
     ⇒ nil
```

The color values are returned for *frame*'s display. If *frame* is omitted or `nil`, the information is returned for the selected frame's display. If the frame cannot display colors, the value is `nil`.

This function used to be called `x-color-values`, and that name is still supported as an alias.

`color-name-to-rgb` *color* **&optional** *frame* [Function]

This function does the same as `color-values`, but it returns color values as floating-point numbers between 0.0 and 1.0 inclusive.

`color-dark-p` *rgb* [Function]

This function returns non-`nil` if the color described by its RGB triplet *rgb* is more readable against white background than against dark background. The argument *rgb* should be a list of the form `(r g b)`, with each component a floating-point number in the range 0.0 to 1.0 inclusive. You can use `color-name-to-rgb` to convert a color's name to such a list.

30.24 Text Terminal Colors

Text terminals usually support only a small number of colors, and the computer uses small integers to select colors on the terminal. This means that the computer cannot reliably tell what the selected color looks like; instead, you have to inform your application which small integers correspond to which colors. However, Emacs does know the standard set of colors and will try to use them automatically.

The functions described in this section control how terminal colors are used by Emacs.

Several of these functions use or return *rgb values*, described in Section 30.23 [Color Names], page 848.

These functions accept a display (either a frame or the name of a terminal) as an optional argument. We hope in the future to make Emacs support different colors on different text terminals; then this argument will specify which terminal to operate on (the default being the selected frame's terminal; see Section 30.10 [Input Focus], page 827). At present, though, the *frame* argument has no effect.

`tty-color-define` *name number* **&optional** *rgb frame* [Function]

This function associates the color name *name* with color number *number* on the terminal.

The optional argument *rgb*, if specified, is an rgb value, a list of three numbers that specify what the color actually looks like. If you do not specify *rgb*, then this color cannot be used by `tty-color-approximate` to approximate other colors, because Emacs will not know what it looks like.

`tty-color-clear` **&optional** *frame* [Function]
This function clears the table of defined colors for a text terminal.

`tty-color-alist` **&optional** *frame* [Function]
This function returns an alist recording the known colors supported by a text terminal.

Each element has the form `(name number . rgb)` or `(name number)`. Here, *name* is the color name, *number* is the number used to specify it to the terminal. If present, *rgb* is a list of three color values (for red, green, and blue) that says what the color actually looks like.

`tty-color-approximate` *rgb* **&optional** *frame* [Function]
This function finds the closest color, among the known colors supported for *display*, to that described by the rgb value *rgb* (a list of color values). The return value is an element of `tty-color-alist`.

`tty-color-translate` *color* **&optional** *frame* [Function]
This function finds the closest color to *color* among the known colors supported for *display* and returns its index (an integer). If the name *color* is not defined, the value is `nil`.

30.25 X Resources

This section describes some of the functions and variables for querying and using X resources, or their equivalent on your operating system. See Section "X Resources" in *The GNU Emacs Manual*, for more information about X resources.

`x-get-resource` *attribute class* **&optional** *component subclass* [Function]
The function `x-get-resource` retrieves a resource value from the X Window defaults database.

Resources are indexed by a combination of a *key* and a *class*. This function searches using a key of the form '*`instance.attribute`*' (where *instance* is the name under which Emacs was invoked), and using '`Emacs.`*`class`*' as the class.

The optional arguments *component* and *subclass* add to the key and the class, respectively. You must specify both of them or neither. If you specify them, the key is '*`instance.component.attribute`*', and the class is '`Emacs.`*`class.subclass`*'.

`x-resource-class` [Variable]
This variable specifies the application name that `x-get-resource` should look up. The default value is `"Emacs"`. You can examine X resources for other application names by binding this variable to some other string, around a call to `x-get-resource`.

`x-resource-name` [Variable]
This variable specifies the instance name that `x-get-resource` should look up. The default value is the name Emacs was invoked with, or the value specified with the '`-name`' or '`-rn`' switches.

To illustrate some of the above, suppose that you have the line:

```
xterm.vt100.background: yellow
```

in your X resources file (whose name is usually `~/.Xdefaults` or `~/.Xresources`). Then:

```
(let ((x-resource-class "XTerm") (x-resource-name "xterm"))
  (x-get-resource "vt100.background" "VT100.Background"))
     ⇒ "yellow"
(let ((x-resource-class "XTerm") (x-resource-name "xterm"))
  (x-get-resource "background" "VT100" "vt100" "Background"))
     ⇒ "yellow"
```

`inhibit-x-resources` [Variable]
: If this variable is non-`nil`, Emacs does not look up X resources, and X resources do not have any effect when creating new frames.

30.26 Display Feature Testing

The functions in this section describe the basic capabilities of a particular display. Lisp programs can use them to adapt their behavior to what the display can do. For example, a program that ordinarily uses a popup menu could use the minibuffer if popup menus are not supported.

The optional argument *display* in these functions specifies which display to ask the question about. It can be a display name, a frame (which designates the display that frame is on), or `nil` (which refers to the selected frame's display, see Section 30.10 [Input Focus], page 827).

See Section 30.23 [Color Names], page 848, Section 30.24 [Text Terminal Colors], page 850, for other functions to obtain information about displays.

`display-popup-menus-p` **&optional** *display* [Function]
: This function returns `t` if popup menus are supported on *display*, `nil` if not. Support for popup menus requires that the mouse be available, since the menu is popped up by clicking the mouse on some portion of the Emacs display.

`display-graphic-p` **&optional** *display* [Function]
: This function returns `t` if *display* is a graphic display capable of displaying several frames and several different fonts at once. This is true for displays that use a window system such as X, and false for text terminals.

`display-mouse-p` **&optional** *display* [Function]
: This function returns `t` if *display* has a mouse available, `nil` if not.

`display-color-p` **&optional** *display* [Function]
: This function returns `t` if the screen is a color screen. It used to be called `x-display-color-p`, and that name is still supported as an alias.

`display-grayscale-p` **&optional** *display* [Function]
: This function returns `t` if the screen can display shades of gray. (All color displays can do this.)

`display-supports-face-attributes-p` *attributes* **&optional** *display* [Function]
: This function returns non-`nil` if all the face attributes in *attributes* are supported (see Section 41.12.1 [Face Attributes], page 1168).

The definition of "supported" is somewhat heuristic, but basically means that a face containing all the attributes in *attributes*, when merged with the default face for display, can be represented in a way that's

1. different in appearance than the default face, and
2. close in spirit to what the attributes specify, if not exact.

Point (2) implies that a `:weight black` attribute will be satisfied by any display that can display bold, as will `:foreground "yellow"` as long as some yellowish color can be displayed, but `:slant italic` will *not* be satisfied by the tty display code's automatic substitution of a dim face for italic.

`display-selections-p` **&optional** *display* [Function]
This function returns `t` if *display* supports selections. Windowed displays normally support selections, but they may also be supported in some other cases.

`display-images-p` **&optional** *display* [Function]
This function returns `t` if *display* can display images. Windowed displays ought in principle to handle images, but some systems lack the support for that. On a display that does not support images, Emacs cannot display a tool bar.

`display-screens` **&optional** *display* [Function]
This function returns the number of screens associated with the display.

`display-pixel-height` **&optional** *display* [Function]
This function returns the height of the screen in pixels. On a character terminal, it gives the height in characters.

For graphical terminals, note that on multi-monitor setups this refers to the pixel height for all physical monitors associated with *display*. See Section 30.2 [Multiple Terminals], page 789.

`display-pixel-width` **&optional** *display* [Function]
This function returns the width of the screen in pixels. On a character terminal, it gives the width in characters.

For graphical terminals, note that on multi-monitor setups this refers to the pixel width for all physical monitors associated with *display*. See Section 30.2 [Multiple Terminals], page 789.

`display-mm-height` **&optional** *display* [Function]
This function returns the height of the screen in millimeters, or `nil` if Emacs cannot get that information.

For graphical terminals, note that on multi-monitor setups this refers to the height for all physical monitors associated with *display*. See Section 30.2 [Multiple Terminals], page 789.

`display-mm-width` **&optional** *display* [Function]
This function returns the width of the screen in millimeters, or `nil` if Emacs cannot get that information.

For graphical terminals, note that on multi-monitor setups this refers to the width for all physical monitors associated with *display*. See Section 30.2 [Multiple Terminals], page 789.

`display-mm-dimensions-alist` [User Option]
: This variable allows the user to specify the dimensions of graphical displays returned by `display-mm-height` and `display-mm-width` in case the system provides incorrect values.

`display-backing-store` **&optional** *display* [Function]
: This function returns the backing store capability of the display. Backing store means recording the pixels of windows (and parts of windows) that are not exposed, so that when exposed they can be displayed very quickly.

  Values can be the symbols `always`, `when-mapped`, or `not-useful`. The function can also return `nil` when the question is inapplicable to a certain kind of display.

`display-save-under` **&optional** *display* [Function]
: This function returns non-`nil` if the display supports the SaveUnder feature. That feature is used by pop-up windows to save the pixels they obscure, so that they can pop down quickly.

`display-planes` **&optional** *display* [Function]
: This function returns the number of planes the display supports. This is typically the number of bits per pixel. For a tty display, it is log to base two of the number of colors supported.

`display-visual-class` **&optional** *display* [Function]
: This function returns the visual class for the screen. The value is one of the symbols `static-gray` (a limited, unchangeable number of grays), `gray-scale` (a full range of grays), `static-color` (a limited, unchangeable number of colors), `pseudo-color` (a limited number of colors), `true-color` (a full range of colors), and `direct-color` (a full range of colors).

`display-color-cells` **&optional** *display* [Function]
: This function returns the number of color cells the screen supports.

These functions obtain additional information about the window system in use where Emacs shows the specified *display*. (Their names begin with `x-` for historical reasons.)

`x-server-version` **&optional** *display* [Function]
: This function returns the list of version numbers of the GUI window system running on *display*, such as the X server on GNU and Unix systems. The value is a list of three integers: the major and minor version numbers of the protocol, and the distributor-specific release number of the window system software itself. On GNU and Unix systems, these are normally the version of the X protocol and the distributor-specific release number of the X server software. On MS-Windows, this is the version of the Windows OS.

`x-server-vendor` **&optional** *display* [Function]
: This function returns the vendor that provided the window system software (as a string). On GNU and Unix systems this really means whoever distributes the X server. On MS-Windows this is the vendor ID string of the Windows OS (Microsoft).

  When the developers of X labeled software distributors as "vendors", they showed their false assumption that no system could ever be developed and distributed non-commercially.

31 Positions

A *position* is the index of a character in the text of a buffer. More precisely, a position identifies the place between two characters (or before the first character, or after the last character), so we can speak of the character before or after a given position. However, we often speak of the character "at" a position, meaning the character after that position.

Positions are usually represented as integers starting from 1, but can also be represented as *markers*—special objects that relocate automatically when text is inserted or deleted so they stay with the surrounding characters. Functions that expect an argument to be a position (an integer), but accept a marker as a substitute, normally ignore which buffer the marker points into; they convert the marker to an integer, and use that integer, exactly as if you had passed the integer as the argument, even if the marker points to the wrong buffer. A marker that points nowhere cannot convert to an integer; using it instead of an integer causes an error. See Chapter 32 [Markers], page 871.

See also the field feature (see Section 33.19.9 [Fields], page 939), which provides functions that are used by many cursor-motion commands.

31.1 Point

Point is a special buffer position used by many editing commands, including the self-inserting typed characters and text insertion functions. Other commands move point through the text to allow editing and insertion at different places.

Like other positions, point designates a place between two characters (or before the first character, or after the last character), rather than a particular character. Usually terminals display the cursor over the character that immediately follows point; point is actually before the character on which the cursor sits.

The value of point is a number no less than 1, and no greater than the buffer size plus 1. If narrowing is in effect (see Section 31.4 [Narrowing], page 867), then point is constrained to fall within the accessible portion of the buffer (possibly at one end of it).

Each buffer has its own value of point, which is independent of the value of point in other buffers. Each window also has a value of point, which is independent of the value of point in other windows on the same buffer. This is why point can have different values in various windows that display the same buffer. When a buffer appears in only one window, the buffer's point and the window's point normally have the same value, so the distinction is rarely important. See Section 29.19 [Window Point], page 761, for more details.

`point` [Function]

This function returns the value of point in the current buffer, as an integer.

```
(point)
     ⇒ 175
```

`point-min` [Function]

This function returns the minimum accessible value of point in the current buffer. This is normally 1, but if narrowing is in effect, it is the position of the start of the region that you narrowed to. (See Section 31.4 [Narrowing], page 867.)

`point-max` [Function]
: This function returns the maximum accessible value of point in the current buffer. This is `(1+ (buffer-size))`, unless narrowing is in effect, in which case it is the position of the end of the region that you narrowed to. (See Section 31.4 [Narrowing], page 867.)

`buffer-end` *flag* [Function]
: This function returns `(point-max)` if *flag* is greater than 0, `(point-min)` otherwise. The argument *flag* must be a number.

`buffer-size` **&optional** *buffer* [Function]
: This function returns the total number of characters in the current buffer. In the absence of any narrowing (see Section 31.4 [Narrowing], page 867), `point-max` returns a value one larger than this.

  If you specify a buffer, *buffer*, then the value is the size of *buffer*.

```
(buffer-size)
     ⇒ 35
(point-max)
     ⇒ 36
```

31.2 Motion

Motion functions change the value of point, either relative to the current value of point, relative to the beginning or end of the buffer, or relative to the edges of the selected window. See Section 31.1 [Point], page 855.

31.2.1 Motion by Characters

These functions move point based on a count of characters. `goto-char` is the fundamental primitive; the other functions use that.

`goto-char` *position* [Command]
: This function sets point in the current buffer to the value *position*.

  If narrowing is in effect, *position* still counts from the beginning of the buffer, but point cannot go outside the accessible portion. If *position* is out of range, `goto-char` moves point to the beginning or the end of the accessible portion.

  When this function is called interactively, *position* is the numeric prefix argument, if provided; otherwise it is read from the minibuffer.

  `goto-char` returns *position*.

`forward-char` **&optional** *count* [Command]
: This function moves point *count* characters forward, towards the end of the buffer (or backward, towards the beginning of the buffer, if *count* is negative). If *count* is `nil`, the default is 1.

  If this attempts to move past the beginning or end of the buffer (or the limits of the accessible portion, when narrowing is in effect), it signals an error with error symbol `beginning-of-buffer` or `end-of-buffer`.

  In an interactive call, *count* is the numeric prefix argument.

`backward-char` **&optional** *count* [Command]
This is just like `forward-char` except that it moves in the opposite direction.

31.2.2 Motion by Words

The functions for parsing words described below use the syntax table and `char-script-table` to decide whether a given character is part of a word. See Chapter 36 [Syntax Tables], page 1025, and see Section 34.6 [Character Properties], page 971.

`forward-word` **&optional** *count* [Command]
This function moves point forward *count* words (or backward if *count* is negative). If *count* is omitted or `nil`, it defaults to 1. In an interactive call, *count* is specified by the numeric prefix argument.

"Moving one word" means moving until point crosses a word-constituent character, which indicates the beginning of a word, and then continue moving until the word ends. By default, characters that begin and end words, known as *word boundaries*, are defined by the current buffer's syntax table (see Section 36.2.1 [Syntax Class Table], page 1026), but modes can override that by setting up a suitable `find-word-boundary-function-table`, described below. Characters that belong to different scripts (as defined by `char-script-table`), also define a word boundary (see Section 34.6 [Character Properties], page 971). In any case, this function cannot move point past the boundary of the accessible portion of the buffer, or across a field boundary (see Section 33.19.9 [Fields], page 939). The most common case of a field boundary is the end of the prompt in the minibuffer.

If it is possible to move *count* words, without being stopped prematurely by the buffer boundary or a field boundary, the value is `t`. Otherwise, the return value is `nil` and point stops at the buffer boundary or field boundary.

If `inhibit-field-text-motion` is non-`nil`, this function ignores field boundaries.

`backward-word` **&optional** *count* [Command]
This function is just like `forward-word`, except that it moves backward until encountering the front of a word, rather than forward.

`words-include-escapes` [User Option]
This variable affects the behavior of `forward-word` and `backward-word`, and everything that uses them. If it is non-`nil`, then characters in the escape and character-quote syntax classes count as part of words. Otherwise, they do not.

`inhibit-field-text-motion` [Variable]
If this variable is non-`nil`, certain motion functions including `forward-word`, `forward-sentence`, and `forward-paragraph` ignore field boundaries.

`find-word-boundary-function-table` [Variable]
This variable affects the behavior of `forward-word` and `backward-word`, and everything that uses them. Its value is a char-table (see Section 6.6 [Char-Tables], page 119) of functions to search for word boundaries. If a character has a non-`nil` entry in this table, then when a word starts or ends with that character, the corresponding function will be called with 2 arguments: *pos* and *limit*. The function should return the position of the other word boundary. Specifically, if *pos* is smaller than *limit*, then

pos is at the beginning of a word, and the function should return the position after the last character of the word; otherwise, *pos* is at the last character of a word, and the function should return the position of that word's first character.

`forward-word-strictly` **&optional** *count* [Function]
This function is like `forward-word`, but it is not affected by `find-word-boundary-function-table`. Lisp programs that should not change behavior when word movement is modified by modes which set that table, such as `subword-mode`, should use this function instead of `forward-word`.

`backward-word-strictly` **&optional** *count* [Function]
This function is like `backward-word`, but it is not affected by `find-word-boundary-function-table`. Like with `forward-word-strictly`, use this function instead of `backward-word` when movement by words should only consider syntax tables.

31.2.3 Motion to an End of the Buffer

To move point to the beginning of the buffer, write:

```
(goto-char (point-min))
```

Likewise, to move to the end of the buffer, use:

```
(goto-char (point-max))
```

Here are two commands that users use to do these things. They are documented here to warn you not to use them in Lisp programs, because they set the mark and display messages in the echo area.

`beginning-of-buffer` **&optional** *n* [Command]
This function moves point to the beginning of the buffer (or the limits of the accessible portion, when narrowing is in effect), setting the mark at the previous position (except in Transient Mark mode, if the mark is already active, it does not set the mark.)

If *n* is non-`nil`, then it puts point *n* tenths of the way from the beginning of the accessible portion of the buffer. In an interactive call, *n* is the numeric prefix argument, if provided; otherwise *n* defaults to `nil`.

Warning: Don't use this function in Lisp programs!

`end-of-buffer` **&optional** *n* [Command]
This function moves point to the end of the buffer (or the limits of the accessible portion, when narrowing is in effect), setting the mark at the previous position (except in Transient Mark mode when the mark is already active). If *n* is non-`nil`, then it puts point *n* tenths of the way from the end of the accessible portion of the buffer.

In an interactive call, *n* is the numeric prefix argument, if provided; otherwise *n* defaults to `nil`.

Warning: Don't use this function in Lisp programs!

31.2.4 Motion by Text Lines

Text lines are portions of the buffer delimited by newline characters, which are regarded as part of the previous line. The first text line begins at the beginning of the buffer, and the last text line ends at the end of the buffer whether or not the last character is a newline.

The division of the buffer into text lines is not affected by the width of the window, by line continuation in display, or by how tabs and control characters are displayed.

`beginning-of-line` **&optional** *count* [Command]
: This function moves point to the beginning of the current line. With an argument *count* not `nil` or 1, it moves forward *count*−1 lines and then to the beginning of the line.

  This function does not move point across a field boundary (see Section 33.19.9 [Fields], page 939) unless doing so would move beyond there to a different line; therefore, if *count* is `nil` or 1, and point starts at a field boundary, point does not move. To ignore field boundaries, either bind `inhibit-field-text-motion` to `t`, or use the `forward-line` function instead. For instance, `(forward-line 0)` does the same thing as `(beginning-of-line)`, except that it ignores field boundaries.

  If this function reaches the end of the buffer (or of the accessible portion, if narrowing is in effect), it positions point there. No error is signaled.

`line-beginning-position` **&optional** *count* [Function]
: Return the position that `(beginning-of-line` *count*`)` would move to.

`end-of-line` **&optional** *count* [Command]
: This function moves point to the end of the current line. With an argument *count* not `nil` or 1, it moves forward *count*−1 lines and then to the end of the line.

  This function does not move point across a field boundary (see Section 33.19.9 [Fields], page 939) unless doing so would move beyond there to a different line; therefore, if *count* is `nil` or 1, and point starts at a field boundary, point does not move. To ignore field boundaries, bind `inhibit-field-text-motion` to `t`.

  If this function reaches the end of the buffer (or of the accessible portion, if narrowing is in effect), it positions point there. No error is signaled.

`line-end-position` **&optional** *count* [Function]
: Return the position that `(end-of-line` *count*`)` would move to.

`pos-bol` **&optional** *count* [Function]
: Like `line-beginning-position`, but ignores fields (and is more efficient).

`pos-eol` **&optional** *count* [Function]
: Like `line-end-position`, but ignores fields (and is more efficient).

`forward-line` **&optional** *count* [Command]
: This function moves point forward *count* lines, to the beginning of the line following that. If *count* is negative, it moves point −*count* lines backward, to the beginning of a line preceding that. If *count* is zero, it moves point to the beginning of the current line. If *count* is `nil`, that means 1.

  If `forward-line` encounters the beginning or end of the buffer (or of the accessible portion) before finding that many lines, it sets point there. No error is signaled.

  `forward-line` returns the difference between *count* and the number of lines actually moved. If you attempt to move down five lines from the beginning of a buffer that has only three lines, point stops at the end of the last line, and the value will be 2.

As an explicit exception, if the last accessible line is non-empty, but has no newline (e.g., if the buffer ends without a newline), the function sets point to the end of that line, and the value returned by the function counts that line as one line successfully moved.

In an interactive call, *count* is the numeric prefix argument.

`count-lines` *start end* **&optional** *ignore-invisible-lines* [Function]

This function returns the number of lines between the positions *start* and *end* in the current buffer. If *start* and *end* are equal, then it returns 0. Otherwise it returns at least 1, even if *start* and *end* are on the same line. This is because the text between them, considered in isolation, must contain at least one line unless it is empty.

If the optional *ignore-invisible-lines* is non-`nil`, invisible lines will not be included in the count.

`count-words` *start end* [Command]

This function returns the number of words between the positions *start* and *end* in the current buffer.

This function can also be called interactively. In that case, it prints a message reporting the number of lines, words, and characters in the buffer, or in the region if the region is active.

`line-number-at-pos` **&optional** *pos absolute* [Function]

This function returns the line number in the current buffer corresponding to the buffer position *pos*. If *pos* is `nil` or omitted, the current buffer position is used. If *absolute* is `nil`, the default, counting starts at `(point-min)`, so the value refers to the contents of the accessible portion of the (potentially narrowed) buffer. If *absolute* is non-`nil`, ignore any narrowing and return the absolute line number.

Also see the functions `bolp` and `eolp` in Section 33.1 [Near Point], page 880. These functions do not move point, but test whether it is already at the beginning or end of a line.

31.2.5 Motion by Screen Lines

The line functions in the previous section count text lines, delimited only by newline characters. By contrast, these functions count screen lines, which are defined by the way the text appears on the screen. A text line is a single screen line if it is short enough to fit the width of the selected window, but otherwise it may occupy several screen lines.

In some cases, text lines are truncated on the screen rather than continued onto additional screen lines. In these cases, `vertical-motion` moves point much like `forward-line`. See Section 41.3 [Truncation], page 1134.

Because the width of a given string depends on the flags that control the appearance of certain characters, `vertical-motion` behaves differently, for a given piece of text, depending on the buffer it is in, and even on the selected window (because the width, the truncation flag, and display table may vary between windows). See Section 41.23.1 [Usual Display], page 1247.

These functions scan text to determine where screen lines break, and thus take time proportional to the distance scanned.

`vertical-motion` *count* **&optional** *window cur-col* [Function]

This function moves point to the start of the screen line *count* screen lines down from the screen line containing point. If *count* is negative, it moves up instead. If *count* is zero, point moves to the visual start of the current screen line.

The *count* argument can be a cons cell, `(cols . lines)`, instead of an integer. Then the function moves by *lines* screen lines, as described for *count* above, and puts point *cols* columns from the visual start of that screen line. The value of *cols* can be a float, and is interpreted in units of the frame's canonical character width (see Section 30.3.2 [Frame Font], page 799); this allows specifying accurate horizontal position of point when the target screen line uses variable fonts. Note that *cols* are counted from the *visual* start of the line; if the window is scrolled horizontally (see Section 29.23 [Horizontal Scrolling], page 770), the column where point will end is in addition to the number of columns by which the text is scrolled, and if the target line is a continuation line, its leftmost column is considered column zero (unlike column-oriented functions, see Section 33.16 [Columns], page 912).

The return value is the number of screen lines over which point was moved. The value may be less in absolute value than *count* if the beginning or end of the buffer was reached.

The window *window* is used for obtaining parameters such as the width, the horizontal scrolling, and the display table. But `vertical-motion` always operates on the current buffer, even if *window* currently displays some other buffer.

The optional argument *cur-col* specifies the current column when the function is called. This is the window-relative horizontal coordinate of point, measured in units of font width of the frame's default face. Providing it speeds up the function, especially in very long lines, because the function doesn't have to go back in the buffer in order to determine the current column. Note that *cur-col* is also counted from the visual start of the line.

`count-screen-lines` **&optional** *beg end count-final-newline window* [Function]

This function returns the number of screen lines in the text from *beg* to *end*. The number of screen lines may be different from the number of actual lines, due to line continuation, the display table, etc. If *beg* and *end* are `nil` or omitted, they default to the beginning and end of the accessible portion of the buffer.

If the region ends with a newline, that is ignored unless the optional third argument *count-final-newline* is non-`nil`.

The optional fourth argument *window* specifies the window for obtaining parameters such as width, horizontal scrolling, and so on. The default is to use the selected window's parameters.

Like `vertical-motion`, `count-screen-lines` always uses the current buffer, regardless of which buffer is displayed in *window*. This makes possible to use `count-screen-lines` in any buffer, whether or not it is currently displayed in some window.

`move-to-window-line` *count* [Command]

This function moves point with respect to the text currently displayed in the selected window. It moves point to the beginning of the screen line *count* screen lines from the top of the window; zero means the topmost line. If *count* is negative, that specifies a

position −*count* lines from the bottom (or the last line of the buffer, if the buffer ends above the specified screen position); thus, *count* of −1 specifies the last fully visible screen line of the window.

If *count* is `nil`, then point moves to the beginning of the line in the middle of the window. If the absolute value of *count* is greater than the size of the window, then point moves to the place that would appear on that screen line if the window were tall enough. This will probably cause the next redisplay to scroll to bring that location onto the screen.

In an interactive call, *count* is the numeric prefix argument.

The value returned is the screen line number point has moved to, relative to the top line of the window.

`move-to-window-group-line` *count* [Function]

This function is like `move-to-window-line`, except that when the selected window is a part of a group of windows (see [Window Group], page 700), `move-to-window-group-line` will move to a position with respect to the entire group, not just the single window. This condition holds when the buffer local variable `move-to-window-group-line-function` is set to a function. In this case, `move-to-window-group-line` calls the function with the argument *count*, then returns its result.

`compute-motion` *from frompos to topos width offsets window* [Function]

This function scans the current buffer, calculating screen positions. It scans the buffer forward from position *from*, assuming that is at screen coordinates *frompos*, to position *to* or coordinates *topos*, whichever comes first. It returns the ending buffer position and screen coordinates.

The coordinate arguments *frompos* and *topos* are cons cells of the form `(`*`hpos`*` . `*`vpos`*`)`.

The argument *width* is the number of columns available to display text; this affects handling of continuation lines. `nil` means the actual number of usable text columns in the window, which is equivalent to the value returned by `(window-width window)`.

The argument *offsets* is either `nil` or a cons cell of the form `(`*`hscroll . tab-offset`*`)`. Here *hscroll* is the number of columns not being displayed at the left margin; most callers get this by calling `window-hscroll`. Meanwhile, *tab-offset* is the offset between column numbers on the screen and column numbers in the buffer. This can be nonzero in a continuation line, when the previous screen lines' widths do not add up to a multiple of `tab-width`. It is always zero in a non-continuation line.

The window *window* serves only to specify which display table to use. `compute-motion` always operates on the current buffer, regardless of what buffer is displayed in *window*.

The return value is a list of five elements:

```
(pos hpos vpos prevhpos contin)
```

Here *pos* is the buffer position where the scan stopped, *vpos* is the vertical screen position, and *hpos* is the horizontal screen position.

The result *prevhpos* is the horizontal position one character back from *pos*. The result *contin* is `t` if the last line was continued after (or within) the previous character.

For example, to find the buffer position of column *col* of screen line *line* of a certain window, pass the window's display start location as *from* and the window's upper-left coordinates as *frompos*. Pass the buffer's `(point-max)` as *to*, to limit the scan to the end of the accessible portion of the buffer, and pass *line* and *col* as *topos*. Here's a function that does this:

```
(defun coordinates-of-position (col line)
  (car (compute-motion (window-start)
                       '(0 . 0)
                       (point-max)
                       (cons col line)
                       (window-width)
                       (cons (window-hscroll) 0)
                       (selected-window))))
```

When you use `compute-motion` for the minibuffer, you need to use `minibuffer-prompt-width` to get the horizontal position of the beginning of the first screen line. See Section 21.12 [Minibuffer Contents], page 421.

31.2.6 Moving over Balanced Expressions

Here are several functions concerned with balanced-parenthesis expressions (also called *sexps* in connection with moving across them in Emacs). The syntax table controls how these functions interpret various characters; see Chapter 36 [Syntax Tables], page 1025. See Section 36.6 [Parsing Expressions], page 1033, for lower-level primitives for scanning sexps or parts of sexps. For user-level commands, see Section "Commands for Editing with Parentheses" in *The GNU Emacs Manual*.

`forward-list` **&optional** *arg* [Command]
: This function moves forward across *arg* (default 1) balanced groups of parentheses. (Other syntactic entities such as words or paired string quotes are ignored.)

`backward-list` **&optional** *arg* [Command]
: This function moves backward across *arg* (default 1) balanced groups of parentheses. (Other syntactic entities such as words or paired string quotes are ignored.)

`up-list` **&optional** *arg escape-strings no-syntax-crossing* [Command]
: This function moves forward out of *arg* (default 1) levels of parentheses. A negative argument means move backward but still to a less deep spot. If *escape-strings* is non-`nil` (as it is interactively), move out of enclosing strings as well. If *no-syntax-crossing* is non-`nil` (as it is interactively), prefer to break out of any enclosing string instead of moving to the start of a list broken across multiple strings. On error, location of point is unspecified.

`backward-up-list` **&optional** *arg escape-strings no-syntax-crossing* [Command]
: This function is just like `up-list`, but with a negated argument.

`down-list` **&optional** *arg* [Command]
: This function moves forward into *arg* (default 1) levels of parentheses. A negative argument means move backward but still go deeper in parentheses ($-$*arg* levels).

`forward-sexp` **&optional** *arg* [Command]
: This function moves forward across *arg* (default 1) balanced expressions. Balanced expressions include both those delimited by parentheses and other kinds, such as words and string constants. See Section 36.6 [Parsing Expressions], page 1033. For example,

```
---------- Buffer: foo ----------
(concat* "foo " (car x) y z)
---------- Buffer: foo ----------

(forward-sexp 3)
     ⇒ nil

---------- Buffer: foo ----------
(concat "foo " (car x) y* z)
---------- Buffer: foo ----------
```

`backward-sexp` **&optional** *arg* [Command]
: This function moves backward across *arg* (default 1) balanced expressions.

`beginning-of-defun` **&optional** *arg* [Command]
: This function moves back to the *arg*th beginning of a defun. If *arg* is negative, this actually moves forward, but it still moves to the beginning of a defun, not to the end of one. *arg* defaults to 1.

`end-of-defun` **&optional** *arg* [Command]
: This function moves forward to the *arg*th end of a defun. If *arg* is negative, this actually moves backward, but it still moves to the end of a defun, not to the beginning of one. *arg* defaults to 1.

`defun-prompt-regexp` [User Option]
: If non-`nil`, this buffer-local variable holds a regular expression that specifies what text can appear before the open-parenthesis that starts a defun. That is to say, a defun begins on a line that starts with a match for this regular expression, followed by a character with open-parenthesis syntax.

`open-paren-in-column-0-is-defun-start` [User Option]
: If this variable's value is non-`nil`, an open parenthesis in column 0 is considered to be the start of a defun. If it is `nil`, an open parenthesis in column 0 has no special meaning. The default is `t`. If a string literal happens to have a parenthesis in column 0, escape it with a backslash to avoid a false positive.

`beginning-of-defun-function` [Variable]
: If non-`nil`, this variable holds a function for finding the beginning of a defun. The function `beginning-of-defun` calls this function instead of using its normal method, passing it its optional argument. If the argument is non-`nil`, the function should move back by that many functions, like `beginning-of-defun` does.

`end-of-defun-function` [Variable]
: If non-`nil`, this variable holds a function for finding the end of a defun. The function `end-of-defun` calls this function instead of using its normal method.

If Emacs is compiled with tree-sitter, it can use the tree-sitter parser information to move across syntax constructs. Since what exactly is considered a defun varies between languages, a major mode should set `treesit-defun-type-regexp` to determine that. Then the mode can get navigation-by-defun functionality for free, by using `treesit-beginning-of-defun` and `treesit-end-of-defun`.

`treesit-defun-type-regexp` [Variable]

This variable determines which nodes are considered defuns by Emacs. It can be a regexp that matches the type of defun nodes. (For "node" and "node type", see Chapter 37 [Parsing Program Source], page 1042.)

For example, `python-mode` sets this variable to a regexp that matches either '`function_definition`' or '`class_definition`'.

Sometimes not all nodes matched by the regexp are valid defuns. Therefore, this variable can also be a cons cell of the form (*regexp* . *pred*), where *pred* should be a function that takes a node as its argument, and returns non-`nil` if the node is a valid defun, or `nil` if it is not valid.

`treesit-defun-tactic` [Variable]

This variable determines how Emacs treats nested defuns. If the value is `top-level`, navigation functions only move across top-level defuns. If the value is `nested`, navigation functions recognize nested defuns.

31.2.7 Skipping Characters

The following two functions move point over a specified set of characters. For example, they are often used to skip whitespace. For related functions, see Section 36.5 [Motion and Syntax], page 1033.

These functions convert the set string to multibyte if the buffer is multibyte, and they convert it to unibyte if the buffer is unibyte, as the search functions do (see Chapter 35 [Searching and Matching], page 996).

`skip-chars-forward` *character-set* **&optional** *limit* [Function]

This function moves point in the current buffer forward, skipping over a given set of characters. It examines the character following point, then advances point if the character matches *character-set*. This continues until it reaches a character that does not match. The function returns the number of characters moved over.

The argument *character-set* is a string, like the inside of a '`[...]`' in a regular expression except that '`]`' does not terminate it, and '`\`' quotes '`^`', '`-`' or '`\`'. Thus, `"a-zA-Z"` skips over all letters, stopping before the first nonletter, and `"^a-zA-Z"` skips nonletters stopping before the first letter (see Section 35.3 [Regular Expressions], page 999). Character classes can also be used, e.g., `"[:alnum:]"` (see Section 35.3.1.2 [Char Classes], page 1003).

If *limit* is supplied (it must be a number or a marker), it specifies the maximum position in the buffer that point can be skipped to. Point will stop at or before *limit*.

In the following example, point is initially located directly before the '`T`'. After the form is evaluated, point is located at the end of that line (between the '`t`' of '`hat`' and the newline). The function skips all letters and spaces, but not newlines.

```
---------- Buffer: foo ----------
I read "⋆The cat in the hat
comes back" twice.
---------- Buffer: foo ----------

(skip-chars-forward "a-zA-Z ")
     ⇒ 18

---------- Buffer: foo ----------
I read "The cat in the hat⋆
comes back" twice.
---------- Buffer: foo ----------
```

`skip-chars-backward` *character-set* **&optional** *limit* [Function]

This function moves point backward, skipping characters that match *character-set*, until *limit*. It is just like `skip-chars-forward` except for the direction of motion.

The return value indicates the distance traveled. It is an integer that is zero or less.

31.3 Excursions

It is often useful to move point temporarily within a localized portion of the program. This is called an *excursion*, and it is done with the `save-excursion` special form. This construct remembers the initial identity of the current buffer, and its value of point, and restores them after the excursion completes. It is the standard way to move point within one part of a program and avoid affecting the rest of the program, and is used thousands of times in the Lisp sources of Emacs.

If you only need to save and restore the identity of the current buffer, use `save-current-buffer` or `with-current-buffer` instead (see Section 28.2 [Current Buffer], page 673). If you need to save or restore window configurations, see the forms described in Section 29.26 [Window Configurations], page 777, and in Section 30.13 [Frame Configurations], page 833.

`save-excursion` *body...* [Special Form]

This special form saves the identity of the current buffer and the value of point in it, evaluates *body*, and finally restores the buffer and its saved value of point. Both saved values are restored even in case of an abnormal exit via `throw` or error (see Section 11.7 [Nonlocal Exits], page 177).

The value returned by `save-excursion` is the result of the last form in *body*, or `nil` if no body forms were given.

Because `save-excursion` only saves point for the buffer that was current at the start of the excursion, any changes made to point in other buffers, during the excursion, will remain in effect afterward. This frequently leads to unintended consequences, so the byte compiler warns if you call `set-buffer` during an excursion:

```
Warning: Use ‘with-current-buffer’ rather than
         save-excursion+set-buffer
```

To avoid such problems, you should call `save-excursion` only after setting the desired current buffer, as in the following example:

```
(defun append-string-to-buffer (string buffer)
  "Append STRING to the end of BUFFER."
  (with-current-buffer buffer
    (save-excursion
      (goto-char (point-max))
      (insert string))))
```

Likewise, `save-excursion` does not restore window-buffer correspondences altered by functions such as `switch-to-buffer`.

Warning: Ordinary insertion of text adjacent to the saved point value relocates the saved value, just as it relocates all markers. More precisely, the saved value is a marker with insertion type `nil`. See Section 32.5 [Marker Insertion Types], page 874. Therefore, when the saved point value is restored, it normally comes before the inserted text.

`save-mark-and-excursion` *body...* [Macro]

This macro is like `save-excursion`, but also saves and restores the mark location and `mark-active`. This macro does what `save-excursion` did before Emacs 25.1.

31.4 Narrowing

Narrowing means limiting the text addressable by Emacs editing commands to a limited range of characters in a buffer. The text that remains addressable is called the *accessible portion* of the buffer.

Narrowing is specified with two buffer positions, which become the beginning and end of the accessible portion. For most editing commands and primitives, these positions replace the values of the beginning and end of the buffer. While narrowing is in effect, no text outside the accessible portion is displayed, and point cannot move outside the accessible portion. Note that narrowing does not alter actual buffer positions (see Section 31.1 [Point], page 855); it only determines which positions are considered the accessible portion of the buffer. Most functions refuse to operate on text that is outside the accessible portion.

Commands for saving buffers are unaffected by narrowing; they save the entire buffer regardless of any narrowing.

If you need to display in a single buffer several very different types of text, consider using an alternative facility described in Section 28.12 [Swapping Text], page 690.

`narrow-to-region` *start end* [Command]

This function sets the accessible portion of the current buffer to start at *start* and end at *end*. Both arguments should be character positions.

In an interactive call, *start* and *end* are set to the bounds of the current region (point and the mark, with the smallest first).

However, when the narrowing has been set by `with-restriction` with a label argument (see below), `narrow-to-region` can be used only within the limits of that narrowing. If *start* or *end* are outside these limits, the corresponding limit set by `with-restriction` is used instead. To gain access to other portions of the buffer, use `without-restriction` with the same label.

`narrow-to-page` **&optional** *move-count* [Command]
: This function sets the accessible portion of the current buffer to include just the current page. An optional first argument *move-count* non-`nil` means to move forward or backward by *move-count* pages and then narrow to one page. The variable `page-delimiter` specifies where pages start and end (see Section 35.8 [Standard Regexps], page 1023).

  In an interactive call, *move-count* is set to the numeric prefix argument.

`widen` [Command]
: This function cancels any narrowing in the current buffer, so that the entire contents are accessible. This is called *widening*. It is equivalent to the following expression:

```
(narrow-to-region 1 (1+ (buffer-size)))
```

  However, when a narrowing has been set by `with-restriction` with a label argument (see below), the limits set by `with-restriction` are restored, instead of canceling the narrowing. To gain access to other portions of the buffer, use `without-restriction` with the same label.

`buffer-narrowed-p` [Function]
: This function returns non-`nil` if the buffer is narrowed, and `nil` otherwise.

`save-restriction` *body...* [Special Form]
: This special form saves the current bounds of the accessible portion, evaluates the *body* forms, and finally restores the saved bounds, thus restoring the same state of narrowing (or absence thereof) formerly in effect. The state of narrowing is restored even in the event of an abnormal exit via `throw` or error (see Section 11.7 [Nonlocal Exits], page 177). Therefore, this construct is a clean way to narrow a buffer temporarily.

  This construct also saves and restores the narrowings that were set by `with-restriction` with a label argument (see below).

  The value returned by `save-restriction` is that returned by the last form in *body*, or `nil` if no body forms were given.

  Caution: it is easy to make a mistake when using the `save-restriction` construct. Read the entire description here before you try it.

  If *body* changes the current buffer, `save-restriction` still restores the restrictions on the original buffer (the buffer whose restrictions it saved from), but it does not restore the identity of the current buffer.

  `save-restriction` does *not* restore point; use `save-excursion` for that. If you use both `save-restriction` and `save-excursion` together, `save-excursion` should come first (on the outside). Otherwise, the old point value would be restored with temporary narrowing still in effect. If the old point value were outside the limits of the temporary narrowing, this would fail to restore it accurately.

  Here is a simple example of correct use of `save-restriction`:

```
---------- Buffer: foo ----------
This is the contents of foo
This is the contents of foo
This is the contents of foo*
---------- Buffer: foo ----------
```

```
(save-excursion
  (save-restriction
    (goto-char 1)
    (forward-line 2)
    (narrow-to-region 1 (point))
    (goto-char (point-min))
    (replace-string "foo" "bar")))

---------- Buffer: foo ----------
This is the contents of bar
This is the contents of bar
This is the contents of foo*
---------- Buffer: foo ----------
```

`with-restriction` *start end* [*:label label*] *body* [Special Form]

This special form saves the current bounds of the accessible portion of the buffer, sets the accessible portion to start at *start* and end at *end*, evaluates the *body* forms, and restores the saved bounds. In that case it is equivalent to

```
(save-restriction
  (narrow-to-region start end)
  body)
```

When the optional argument *label*, which is evaluated to get the label to use and must yield a non-`nil` value, is present, the narrowing is *labeled*. A labeled narrowing differs from a non-labeled one in several ways:

- During the evaluation of the *body* form, `narrow-to-region` and `widen` can be used only within the *start* and *end* limits.
- To lift the restriction introduced by `with-restriction` and gain access to other portions of the buffer, use `without-restriction` with the same *label* argument. (Another way to gain access to other portions of the buffer is to use an indirect buffer (see Section 28.11 [Indirect Buffers], page 689).)
- Labeled narrowings can be nested.
- Labeled narrowings can only be used in Lisp programs: they are never visible on display, and never interfere with narrowings set by the user.

If you use `with-restriction` with the optional *label* argument, we recommend documenting the *label* in the doc strings of the functions which use it, so that other Lisp programs your code calls could lift the labeled narrowing if and when it needs.

`without-restriction` [*:label label*] *body* [Special Form]

This special form saves the current bounds of the accessible portion of the buffer, widens the buffer, evaluates the *body* forms, and restores the saved bounds. In that case it is equivalent to

```
(save-restriction
  (widen)
  body)
```

When the optional argument *label* is present, the narrowing set by `with-restriction` with the same *label* argument is lifted.

32 Markers

A *marker* is a Lisp object used to specify a position in a buffer relative to the surrounding text. A marker changes its offset from the beginning of the buffer automatically whenever text is inserted or deleted, so that it stays with the two characters on either side of it.

32.1 Overview of Markers

A marker specifies a buffer and a position in that buffer. A marker can be used to represent a position in functions that require one, just as an integer could be used. In that case, the marker's buffer is normally ignored. Of course, a marker used in this way usually points to a position in the buffer that the function operates on, but that is entirely the programmer's responsibility. See Chapter 31 [Positions], page 855, for a complete description of positions.

A marker has three attributes: the marker position, the marker buffer, and the insertion type. The marker position is an integer that is equivalent (at a given time) to the marker as a position in that buffer. But the marker's position value can change during the life of the marker, and often does. Insertion and deletion of text in the buffer relocate the marker. The idea is that a marker positioned between two characters remains between those two characters despite insertion and deletion elsewhere in the buffer. Relocation changes the integer equivalent of the marker.

Deleting text around a marker's position leaves the marker between the characters immediately before and after the deleted text. Inserting text at the position of a marker normally leaves the marker either in front of or after the new text, depending on the marker's *insertion type* (see Section 32.5 [Marker Insertion Types], page 874)—unless the insertion is done with `insert-before-markers` (see Section 33.4 [Insertion], page 884).

Insertion and deletion in a buffer must check all the markers and relocate them if necessary. This slows processing in a buffer with a large number of markers. For this reason, it is a good idea to make a marker point nowhere if you are sure you don't need it any more. Markers that can no longer be accessed are eventually removed (see Section E.3 [Garbage Collection], page 1355).

Because it is common to perform arithmetic operations on a marker position, most of these operations (including `+` and `-`) accept markers as arguments. In such cases, the marker stands for its current position.

Here are examples of creating markers, setting markers, and moving point to markers:

```
;; Make a new marker that initially does not point anywhere:
(setq m1 (make-marker))
     ⇒ #<marker in no buffer>

;; Set m1 to point between the 99th and 100th characters
;;    in the current buffer:
(set-marker m1 100)
     ⇒ #<marker at 100 in markers.texi>
```

```
;; Now insert one character at the beginning of the buffer:
(goto-char (point-min))
     ⇒ 1
(insert "Q")
     ⇒ nil

;; m1 is updated appropriately.
m1
     ⇒ #<marker at 101 in markers.texi>

;; Two markers that point to the same position
;;    are not eq, but they are equal.
(setq m2 (copy-marker m1))
     ⇒ #<marker at 101 in markers.texi>
(eq m1 m2)
     ⇒ nil
(equal m1 m2)
     ⇒ t

;; When you are finished using a marker, make it point nowhere.
(set-marker m1 nil)
     ⇒ #<marker in no buffer>
```

32.2 Predicates on Markers

You can test an object to see whether it is a marker, or whether it is either an integer or a marker. The latter test is useful in connection with the arithmetic functions that work with both markers and integers.

`markerp` *object* [Function]
: This function returns `t` if *object* is a marker, `nil` otherwise. Note that integers are not markers, even though many functions will accept either a marker or an integer.

`integer-or-marker-p` *object* [Function]
: This function returns `t` if *object* is an integer or a marker, `nil` otherwise.

`number-or-marker-p` *object* [Function]
: This function returns `t` if *object* is a number (either integer or floating point) or a marker, `nil` otherwise.

32.3 Functions that Create Markers

When you create a new marker, you can make it point nowhere, or point to the present position of point, or to the beginning or end of the accessible portion of the buffer, or to the same place as another given marker.

The next four functions all return markers with insertion type `nil`. See Section 32.5 [Marker Insertion Types], page 874.

`make-marker` [Function]
This function returns a newly created marker that does not point anywhere.

```
(make-marker)
     ⇒ #<marker in no buffer>
```

`point-marker` [Function]
This function returns a new marker that points to the present position of point in the current buffer. See Section 31.1 [Point], page 855. For an example, see `copy-marker`, below.

`point-min-marker` [Function]
This function returns a new marker that points to the beginning of the accessible portion of the buffer. This will be the beginning of the buffer unless narrowing is in effect. See Section 31.4 [Narrowing], page 867.

`point-max-marker` [Function]
This function returns a new marker that points to the end of the accessible portion of the buffer. This will be the end of the buffer unless narrowing is in effect. See Section 31.4 [Narrowing], page 867.

Here are examples of this function and `point-min-marker`, shown in a buffer containing a version of the source file for the text of this chapter.

```
(point-min-marker)
     ⇒ #<marker at 1 in markers.texi>
(point-max-marker)
     ⇒ #<marker at 24080 in markers.texi>

(narrow-to-region 100 200)
     ⇒ nil
(point-min-marker)
     ⇒ #<marker at 100 in markers.texi>
(point-max-marker)
     ⇒ #<marker at 200 in markers.texi>
```

`copy-marker` **&optional** *marker-or-integer insertion-type* [Function]
If passed a marker as its argument, `copy-marker` returns a new marker that points to the same place and the same buffer as does *marker-or-integer*. If passed an integer as its argument, `copy-marker` returns a new marker that points to position *marker-or-integer* in the current buffer.

The new marker's insertion type is specified by the argument *insertion-type*. See Section 32.5 [Marker Insertion Types], page 874.

```
(copy-marker 0)
     ⇒ #<marker at 1 in markers.texi>

(copy-marker 90000)
     ⇒ #<marker at 24080 in markers.texi>
```

An error is signaled if *marker* is neither a marker nor an integer.

Two distinct markers are considered `equal` (even though not `eq`) to each other if they have the same position and buffer, or if they both point nowhere.

```
(setq p (point-marker))
     ⇒ #<marker at 2139 in markers.texi>

(setq q (copy-marker p))
     ⇒ #<marker at 2139 in markers.texi>

(eq p q)
     ⇒ nil

(equal p q)
     ⇒ t
```

32.4 Information from Markers

This section describes the functions for accessing the components of a marker object.

`marker-position` *marker* [Function]
: This function returns the position that *marker* points to, or `nil` if it points nowhere.

`marker-buffer` *marker* [Function]
: This function returns the buffer that *marker* points into, or `nil` if it points nowhere.

```
(setq m (make-marker))
     ⇒ #<marker in no buffer>
(marker-position m)
     ⇒ nil
(marker-buffer m)
     ⇒ nil

(set-marker m 3770 (current-buffer))
     ⇒ #<marker at 3770 in markers.texi>
(marker-buffer m)
     ⇒ #<buffer markers.texi>
(marker-position m)
     ⇒ 3770
```

32.5 Marker Insertion Types

When you insert text directly at the place where a marker points, there are two possible ways to relocate that marker: it can point before the inserted text, or point after it. You can specify which one a given marker should do by setting its *insertion type*. Note that use of `insert-before-markers` ignores markers' insertion types, always relocating a marker to point after the inserted text.

`set-marker-insertion-type` *marker type* [Function]
: This function sets the insertion type of marker *marker* to *type*. If *type* is `t`, *marker* will advance when text is inserted at its position. If *type* is `nil`, *marker* does not advance when text is inserted there.

`marker-insertion-type` *marker* [Function]
This function reports the current insertion type of *marker*.

All functions that create markers without accepting an argument that specifies the insertion type, create them with insertion type `nil` (see Section 32.3 [Creating Markers], page 872). Also, the mark has, by default, insertion type `nil`.

32.6 Moving Marker Positions

This section describes how to change the position of an existing marker. When you do this, be sure you know whether the marker is used outside of your program, and, if so, what effects will result from moving it—otherwise, confusing things may happen in other parts of Emacs.

`set-marker` *marker position* **&optional** *buffer* [Function]
This function moves *marker* to *position* in *buffer*. If *buffer* is not provided, it defaults to the current buffer.

If *position* is `nil` or a marker that points nowhere, then *marker* is set to point nowhere.

The value returned is *marker*.

```
(setq m (point-marker))
     ⇒ #<marker at 4714 in markers.texi>
(set-marker m 55)
     ⇒ #<marker at 55 in markers.texi>
(setq b (get-buffer "foo"))
     ⇒ #<buffer foo>
(set-marker m 0 b)
     ⇒ #<marker at 1 in foo>
```

`move-marker` *marker position* **&optional** *buffer* [Function]
This is another name for `set-marker`.

32.7 The Mark

Each buffer has a special marker, which is designated *the mark*. When a buffer is newly created, this marker exists but does not point anywhere; this means that the mark doesn't exist in that buffer yet. Subsequent commands can set the mark.

The mark specifies a position to bound a range of text for many commands, such as `kill-region` and `indent-rigidly`. These commands typically act on the text between point and the mark, which is called the *region*. If you are writing a command that operates on the region, don't examine the mark directly; instead, use `interactive` with the '`r`' specification. This provides the values of point and the mark as arguments to the command in an interactive call, but permits other Lisp programs to specify arguments explicitly. See Section 22.2.2 [Interactive Codes], page 428.

Some commands set the mark as a side-effect. Commands should do this only if it has a potential use to the user, and never for their own internal purposes. For example, the `replace-regexp` command sets the mark to the value of point before doing any replacements, because this enables the user to move back there conveniently after the replace is finished.

Once the mark exists in a buffer, it normally never ceases to exist. However, it may become *inactive*, if Transient Mark mode is enabled. The buffer-local variable `mark-active`, if non-`nil`, means that the mark is active. A command can call the function `deactivate-mark` to deactivate the mark directly, or it can request deactivation of the mark upon return to the editor command loop by setting the variable `deactivate-mark` to a non-`nil` value.

If Transient Mark mode is enabled, certain editing commands that normally apply to text near point, apply instead to the region when the mark is active. This is the main motivation for using Transient Mark mode. (Another is that this enables highlighting of the region when the mark is active. See Chapter 41 [Display], page 1133.)

In addition to the mark, each buffer has a *mark ring* which is a list of markers containing previous values of the mark. When editing commands change the mark, they should normally save the old value of the mark on the mark ring. The variable `mark-ring-max` specifies the maximum number of entries in the mark ring; once the list becomes this long, adding a new element deletes the last element.

There is also a separate global mark ring, but that is used only in a few particular user-level commands, and is not relevant to Lisp programming. So we do not describe it here.

`mark` **&optional** *force* [Function]

This function returns the current buffer's mark position as an integer, or `nil` if no mark has ever been set in this buffer.

If Transient Mark mode is enabled, and `mark-even-if-inactive` is `nil`, `mark` signals an error if the mark is inactive. However, if *force* is non-`nil`, then `mark` disregards inactivity of the mark, and returns the mark position (or `nil`) anyway.

`mark-marker` [Function]

This function returns the marker that represents the current buffer's mark. It is not a copy, it is the marker used internally. Therefore, changing this marker's position will directly affect the buffer's mark. Don't do that unless that is the effect you want.

```
(setq m (mark-marker))
     ⇒ #<marker at 3420 in markers.texi>
(set-marker m 100)
     ⇒ #<marker at 100 in markers.texi>
(mark-marker)
     ⇒ #<marker at 100 in markers.texi>
```

Like any marker, this marker can be set to point at any buffer you like. If you make it point at any buffer other than the one of which it is the mark, it will yield perfectly consistent, but rather odd, results. We recommend that you not do it!

`set-mark` *position* [Function]

This function sets the mark to *position*, and activates the mark. The old value of the mark is *not* pushed onto the mark ring.

Please note: Use this function only if you want the user to see that the mark has moved, and you want the previous mark position to be lost. Normally, when a new mark is set, the old one should go on the `mark-ring`. For this reason, most applications should use `push-mark` and `pop-mark`, not `set-mark`.

Novice Emacs Lisp programmers often try to use the mark for the wrong purposes. The mark saves a location for the user's convenience. An editing command should not alter the mark unless altering the mark is part of the user-level functionality of the command. (And, in that case, this effect should be documented.) To remember a location for internal use in the Lisp program, store it in a Lisp variable. For example:

```
(let ((beg (point)))
  (forward-line 1)
  (delete-region beg (point))).
```

`push-mark` **&optional** *position nomsg activate* [Function]

This function sets the current buffer's mark to *position*, and pushes a copy of the previous mark onto `mark-ring`. If *position* is `nil`, then the value of point is used.

The function `push-mark` normally *does not* activate the mark. To do that, specify `t` for the argument *activate*.

A '`Mark set`' message is displayed unless *nomsg* is non-`nil`.

`pop-mark` [Function]

This function pops off the top element of `mark-ring` and makes that mark become the buffer's actual mark. This does not move point in the buffer, and it does nothing if `mark-ring` is empty. It deactivates the mark.

`transient-mark-mode` [User Option]

This variable, if non-`nil`, enables Transient Mark mode. In Transient Mark mode, every buffer-modifying primitive sets `deactivate-mark`. As a consequence, most commands that modify the buffer also deactivate the mark.

When Transient Mark mode is enabled and the mark is active, many commands that normally apply to the text near point instead apply to the region. Such commands should use the function `use-region-p` to test whether they should operate on the region. See Section 32.8 [The Region], page 879.

Lisp programs can set `transient-mark-mode` to non-`nil`, non-`t` values to enable Transient Mark mode temporarily. If the value is `lambda`, Transient Mark mode is automatically turned off after any action, such as buffer modification, that would normally deactivate the mark. If the value is `(only . oldval)`, then `transient-mark-mode` is set to the value *oldval* after any subsequent command that moves point and is not shift-translated (see Section 22.8.1 [Key Sequence Input], page 462), or after any other action that would normally deactivate the mark. (Marking a region with the mouse will temporarily enable `transient-mark-mode` in this way.)

`mark-even-if-inactive` [User Option]

If this is non-`nil`, Lisp programs and the Emacs user can use the mark even when it is inactive. This option affects the behavior of Transient Mark mode. When the option is non-`nil`, deactivation of the mark turns off region highlighting, but commands that use the mark behave as if the mark were still active.

`deactivate-mark` [Variable]

If an editor command sets this variable non-`nil`, then the editor command loop deactivates the mark after the command returns (if Transient Mark mode is enabled).

All the primitives that change the buffer set `deactivate-mark`, to deactivate the mark when the command is finished. Setting this variable makes it buffer-local.

To write Lisp code that modifies the buffer without causing deactivation of the mark at the end of the command, bind `deactivate-mark` to `nil` around the code that does the modification. For example:

```
(let (deactivate-mark)
  (insert " "))
```

`deactivate-mark` **&optional** *force* [Function]
If Transient Mark mode is enabled or *force* is non-`nil`, this function deactivates the mark and runs the normal hook `deactivate-mark-hook`. Otherwise, it does nothing.

`mark-active` [Variable]
The mark is active when this variable is non-`nil`. This variable is always buffer-local in each buffer. Do *not* use the value of this variable to decide whether a command that normally operates on text near point should operate on the region instead. Use the function `use-region-p` for that (see Section 32.8 [The Region], page 879).

`activate-mark-hook` [Variable]
`deactivate-mark-hook` [Variable]
These normal hooks are run, respectively, when the mark becomes active and when it becomes inactive. The hook `activate-mark-hook` is also run when the region is reactivated, for instance after using a command that switches back to a buffer that has an active mark.

`handle-shift-selection` [Function]
This function implements the shift-selection behavior of point-motion commands. See Section "Shift Selection" in *The GNU Emacs Manual*. It is called automatically by the Emacs command loop whenever a command with a '`^`' character in its `interactive` spec is invoked, before the command itself is executed (see Section 22.2.2 [Interactive Codes], page 428).

If `shift-select-mode` is non-`nil` and the current command was invoked via shift translation (see Section 22.8.1 [Key Sequence Input], page 462), this function sets the mark and temporarily activates the region, unless the region was already temporarily activated in this way. Otherwise, if the region has been activated temporarily, it deactivates the mark and restores the variable `transient-mark-mode` to its earlier value.

`mark-ring` [Variable]
The value of this buffer-local variable is the list of saved former marks of the current buffer, most recent first.

```
mark-ring
⇒ (#<marker at 11050 in markers.texi>
    #<marker at 10832 in markers.texi>
    ...)
```

`mark-ring-max` [User Option]
: The value of this variable is the maximum size of `mark-ring`. If more marks than this are pushed onto the `mark-ring`, `push-mark` discards an old mark when it adds a new one.

When Delete Selection mode (see Section "Using Region" in *The GNU Emacs Manual*) is enabled, commands that operate on the active region (a.k.a. "selection") behave slightly differently. This works by adding the function `delete-selection-pre-hook` to the `pre-command-hook` (see Section 22.1 [Command Overview], page 424). That function calls `delete-selection-helper` to delete the selection as appropriate for the command. If you want to adapt a command to Delete Selection mode, put the `delete-selection` property on the function's symbol (see Section 9.4.1 [Symbol Plists], page 138); commands that don't have this property on their symbol won't delete the selection. This property can have one of several values to tailor the behavior to what the command is supposed to do; see the doc strings of `delete-selection-pre-hook` and `delete-selection-helper` for the details.

32.8 The Region

The text between point and the mark is known as *the region*. Various functions operate on text delimited by point and the mark, but only those functions specifically related to the region itself are described here.

The next two functions signal an error if the mark does not point anywhere. If Transient Mark mode is enabled and `mark-even-if-inactive` is `nil`, they also signal an error if the mark is inactive.

`region-beginning` [Function]
: This function returns the position of the beginning of the region (as an integer). This is the position of either point or the mark, whichever is smaller.

`region-end` [Function]
: This function returns the position of the end of the region (as an integer). This is the position of either point or the mark, whichever is larger.

Instead of using `region-beginning` and `region-end`, a command designed to operate on a region should normally use `interactive` with the '`r`' specification to find the beginning and end of the region. This lets other Lisp programs specify the bounds explicitly as arguments. See Section 22.2.2 [Interactive Codes], page 428.

`use-region-p` [Function]
: This function returns `t` if Transient Mark mode is enabled, the mark is active, and there is a valid region in the buffer. This function is intended to be used by commands that operate on the region, instead of on text near point, when the mark is active.

  A region is valid if it has a non-zero size, or if the user option `use-empty-active-region` is non-`nil` (by default, it is `nil`). The function `region-active-p` is similar to `use-region-p`, but considers all regions as valid. In most cases, you should not use `region-active-p`, since if the region is empty it is often more appropriate to operate on point.

33 Text

This chapter describes the functions that deal with the text in a buffer. Most examine, insert, or delete text in the current buffer, often operating at point or on text adjacent to point. Many are interactive. All the functions that change the text provide for undoing the changes (see Section 33.9 [Undo], page 898).

Many text-related functions operate on a region of text defined by two buffer positions passed in arguments named *start* and *end*. These arguments should be either markers (see Chapter 32 [Markers], page 871) or numeric character positions (see Chapter 31 [Positions], page 855). The order of these arguments does not matter; it is all right for *start* to be the end of the region and *end* the beginning. For example, `(delete-region 1 10)` and `(delete-region 10 1)` are equivalent. An `args-out-of-range` error is signaled if either *start* or *end* is outside the accessible portion of the buffer. In an interactive call, point and the mark are used for these arguments.

Throughout this chapter, "text" refers to the characters in the buffer, together with their properties (when relevant). Keep in mind that point is always between two characters, and the cursor appears on the character after point.

33.1 Examining Text Near Point

Many functions are provided to look at the characters around point. Several simple functions are described here. See also `looking-at` in Section 35.4 [Regexp Search], page 1011.

In the following four functions, "beginning" or "end" of buffer refers to the beginning or end of the accessible portion.

`char-after` **&optional** *position* [Function]

This function returns the character in the current buffer at (i.e., immediately after) position *position*. If *position* is out of range for this purpose, either before the beginning of the buffer, or at or beyond the end, then the value is `nil`. The default for *position* is point.

In the following example, assume that the first character in the buffer is '`@`':

```
(string (char-after 1))
     ⇒ "@"
```

`char-before` **&optional** *position* [Function]

This function returns the character in the current buffer immediately before position *position*. If *position* is out of range for this purpose, either at or before the beginning of the buffer, or beyond the end, then the value is `nil`. The default for *position* is point.

`following-char` [Function]

This function returns the character following point in the current buffer. This is similar to `(char-after (point))`. However, if point is at the end of the buffer, then `following-char` returns 0.

Remember that point is always between characters, and the cursor normally appears over the character following point. Therefore, the character returned by `following-char` is the character the cursor is over.

In this example, point is between the 'a' and the 'c'.

```
---------- Buffer: foo ----------
Gentlemen may cry ``Pea*ce! Peace!,''
but there is no peace.
---------- Buffer: foo ----------

(string (preceding-char))
     ⇒ "a"
(string (following-char))
     ⇒ "c"
```

`preceding-char` [Function]

This function returns the character preceding point in the current buffer. See above, under `following-char`, for an example. If point is at the beginning of the buffer, `preceding-char` returns 0.

`bobp` [Function]

This function returns `t` if point is at the beginning of the buffer. If narrowing is in effect, this means the beginning of the accessible portion of the text. See also `point-min` in Section 31.1 [Point], page 855.

`eobp` [Function]

This function returns `t` if point is at the end of the buffer. If narrowing is in effect, this means the end of accessible portion of the text. See also `point-max` in See Section 31.1 [Point], page 855.

`bolp` [Function]

This function returns `t` if point is at the beginning of a line. See Section 31.2.4 [Text Lines], page 858. The beginning of the buffer (or of its accessible portion) always counts as the beginning of a line.

`eolp` [Function]

This function returns `t` if point is at the end of a line. The end of the buffer (or of its accessible portion) is always considered the end of a line.

33.2 Examining Buffer Contents

This section describes functions that allow a Lisp program to convert any portion of the text in the buffer into a string.

`buffer-substring` *start end* [Function]

This function returns a string containing a copy of the text of the region defined by positions *start* and *end* in the current buffer. If the arguments are not positions in the accessible portion of the buffer, `buffer-substring` signals an `args-out-of-range` error.

Here's an example which assumes Font-Lock mode is not enabled:

```
---------- Buffer: foo ----------
This is the contents of buffer foo

---------- Buffer: foo ----------
```

```
(buffer-substring 1 10)
     ⇒ "This is t"
(buffer-substring (point-max) 10)
     ⇒ "he contents of buffer foo\n"
```

If the text being copied has any text properties, these are copied into the string along with the characters they belong to. See Section 33.19 [Text Properties], page 919. However, overlays (see Section 41.9 [Overlays], page 1154) in the buffer and their properties are ignored, not copied.

For example, if Font-Lock mode is enabled, you might get results like these:

```
(buffer-substring 1 10)
     ⇒ #("This is t" 0 1 (fontified t) 1 9 (fontified t))
```

`buffer-substring-no-properties` *start end* [Function]
This is like `buffer-substring`, except that it does not copy text properties, just the characters themselves. See Section 33.19 [Text Properties], page 919.

`buffer-string` [Function]
This function returns the contents of the entire accessible portion of the current buffer, as a string. If the text being copied has any text properties, these are copied into the string along with the characters they belong to.

If you need to make sure the resulting string, when copied to a different location, will not change its visual appearance due to reordering of bidirectional text, use the `buffer-substring-with-bidi-context` function (see Section 41.27 [Bidirectional Display], page 1255).

`filter-buffer-substring` *start end* **&optional** *delete* [Function]
This function filters the buffer text between *start* and *end* using a function specified by the variable `filter-buffer-substring-function`, and returns the result.

The default filter function consults the obsolete wrapper hook `filter-buffer-substring-functions` (see the documentation string of the macro `with-wrapper-hook` for the details about this obsolete facility). If it is `nil`, it returns the unaltered text from the buffer, i.e., what `buffer-substring` would return.

If *delete* is non-`nil`, the function deletes the text between *start* and *end* after copying it, like `delete-and-extract-region`.

Lisp code should use this function instead of `buffer-substring`, `buffer-substring-no-properties`, or `delete-and-extract-region` when copying into user-accessible data structures such as the kill-ring, X clipboard, and registers. Major and minor modes can modify `filter-buffer-substring-function` to alter such text as it is copied out of the buffer.

`filter-buffer-substring-function` [Variable]
The value of this variable is a function that `filter-buffer-substring` will call to do the actual work. The function receives three arguments, the same as those of `filter-buffer-substring`, which it should treat as per the documentation of that function. It should return the filtered text (and optionally delete the source text).

The following two variables are obsoleted by `filter-buffer-substring-function`, but are still supported for backward compatibility.

`filter-buffer-substring-functions` [Variable]

This obsolete variable is a wrapper hook, whose members should be functions that accept four arguments: *fun*, *start*, *end*, and *delete*. *fun* is a function that takes three arguments (*start*, *end*, and *delete*), and returns a string. In both cases, the *start*, *end*, and *delete* arguments are the same as those of `filter-buffer-substring`.

The first hook function is passed a *fun* that is equivalent to the default operation of `filter-buffer-substring`, i.e., it returns the buffer-substring between *start* and *end* and optionally deletes the original text from the buffer. In most cases, the hook function will call *fun* once, and then do its own processing of the result. The next hook function receives a *fun* equivalent to this, and so on. The actual return value is the result of all the hook functions acting in sequence.

`current-word` **&optional** *strict really-word* [Function]

This function returns the symbol (or word) at or near point, as a string. The return value includes no text properties.

If the optional argument *really-word* is non-`nil`, it finds a word; otherwise, it finds a symbol (which includes both word characters and symbol constituent characters).

If the optional argument *strict* is non-`nil`, then point must be in or next to the symbol or word—if no symbol or word is there, the function returns `nil`. Otherwise, a nearby symbol or word on the same line is acceptable.

`thing-at-point` *thing* **&optional** *no-properties* [Function]

Return the *thing* around or next to point, as a string.

The argument *thing* is a symbol which specifies a kind of syntactic entity. Possibilities include `symbol`, `list`, `sexp`, `defun`, `filename`, `existing-filename`, `url`, `word`, `sentence`, `whitespace`, `line`, `page`, `string`, and others.

When the optional argument *no-properties* is non-`nil`, this function strips text properties from the return value.

```
---------- Buffer: foo ----------
Gentlemen may cry ``Pea*ce! Peace!,''
but there is no peace.
---------- Buffer: foo ----------

(thing-at-point 'word)
     ⇒ "Peace"
(thing-at-point 'line)
     ⇒ "Gentlemen may cry ``Peace! Peace!,''\n"
(thing-at-point 'whitespace)
     ⇒ nil
```

`thing-at-point-provider-alist` [Variable]

This variable allows users and modes to tweak how `thing-at-point` works. It's an association list of *thing*s and functions (called with zero parameters) to

return that thing. Entries for *thing* will be evaluated in turn until a non-`nil` result is returned.

For instance, a major mode could say:

```
(setq-local thing-at-point-provider-alist
            (append thing-at-point-provider-alist
                    '((url . my-mode--url-at-point))))
```

If no providers have a non-`nil` return, the *thing* will be computed the standard way.

33.3 Comparing Text

This function lets you compare portions of the text in a buffer, without copying them into strings first.

`compare-buffer-substrings` *buffer1 start1 end1 buffer2 start2 end2* [Function]

This function lets you compare two substrings of the same buffer or two different buffers. The first three arguments specify one substring, giving a buffer (or a buffer name) and two positions within the buffer. The last three arguments specify the other substring in the same way. You can use `nil` for *buffer1*, *buffer2*, or both to stand for the current buffer.

The value is negative if the first substring is less, positive if the first is greater, and zero if they are equal. The absolute value of the result is one plus the index of the first differing characters within the substrings.

This function ignores case when comparing characters if `case-fold-search` is non-`nil`. It always ignores text properties.

Suppose you have the text '`foobarbar haha!rara!`' in the current buffer; then in this example the two substrings are '`rbar `' and '`rara!`'. The value is 2 because the first substring is greater at the second character.

```
(compare-buffer-substrings nil 6 11 nil 16 21)
     ⇒ 2
```

33.4 Inserting Text

Insertion means adding new text to a buffer. The inserted text goes at point—between the character before point and the character after point. Some insertion functions leave point before the inserted text, while other functions leave it after. We call the former insertion *after point* and the latter insertion *before point*.

Insertion moves markers located at positions after the insertion point, so that they stay with the surrounding text (see Chapter 32 [Markers], page 871). When a marker points at the place of insertion, insertion may or may not relocate the marker, depending on the marker's insertion type (see Section 32.5 [Marker Insertion Types], page 874). Certain special functions such as `insert-before-markers` relocate all such markers to point after the inserted text, regardless of the markers' insertion type.

Insertion functions signal an error if the current buffer is read-only (see Section 28.7 [Read Only Buffers], page 682) or if they insert within read-only text (see Section 33.19.4 [Special Properties], page 927).

These functions copy text characters from strings and buffers along with their properties. The inserted characters have exactly the same properties as the characters they were copied from. By contrast, characters specified as separate arguments, not part of a string or buffer, inherit their text properties from the neighboring text.

The insertion functions convert text from unibyte to multibyte in order to insert in a multibyte buffer, and vice versa—if the text comes from a string or from a buffer. However, they do not convert unibyte character codes 128 through 255 to multibyte characters, not even if the current buffer is a multibyte buffer. See Section 34.3 [Converting Representations], page 968.

`insert` **&rest** *args* [Function]

This function inserts the strings and/or characters *args* into the current buffer, at point, moving point forward. In other words, it inserts the text before point. An error is signaled unless all *args* are either strings or characters. The value is `nil`.

`insert-before-markers` **&rest** *args* [Function]

This function inserts the strings and/or characters *args* into the current buffer, at point, moving point forward. An error is signaled unless all *args* are either strings or characters. The value is `nil`.

This function is unlike the other insertion functions in that it relocates markers initially pointing at the insertion point, to point after the inserted text. If an overlay begins at the insertion point, the inserted text falls outside the overlay; if a nonempty overlay ends at the insertion point, the inserted text falls inside that overlay.

`insert-char` *character* **&optional** *count inherit* [Command]

This command inserts *count* instances of *character* into the current buffer before point. The argument *count* must be an integer, and *character* must be a character.

If called interactively, this command prompts for *character* using its Unicode name or its code point. See Section "Inserting Text" in *The GNU Emacs Manual*.

This function does not convert unibyte character codes 128 through 255 to multibyte characters, not even if the current buffer is a multibyte buffer. See Section 34.3 [Converting Representations], page 968.

If *inherit* is non-`nil`, the inserted characters inherit sticky text properties from the two characters before and after the insertion point. See Section 33.19.6 [Sticky Properties], page 935.

`insert-buffer-substring` *from-buffer-or-name* **&optional** *start end* [Function]

This function inserts a portion of buffer *from-buffer-or-name* into the current buffer before point. The text inserted is the region between *start* (inclusive) and *end* (exclusive). (These arguments default to the beginning and end of the accessible portion of that buffer.) This function returns `nil`.

In this example, the form is executed with buffer '`bar`' as the current buffer. We assume that buffer '`bar`' is initially empty.

```
---------- Buffer: foo ----------
We hold these truths to be self-evident, that all
---------- Buffer: foo ----------
```

```
(insert-buffer-substring "foo" 1 20)
     ⇒ nil

---------- Buffer: bar ----------
We hold these truth*
---------- Buffer: bar ----------
```

`insert-buffer-substring-no-properties` *from-buffer-or-name* **&optional** *start end* [Function]

This is like `insert-buffer-substring` except that it does not copy any text properties.

`insert-into-buffer` *to-buffer* **&optional** *start end* [Function]

This is like `insert-buffer-substring`, but works in the opposite direction: The text is copied from the current buffer into *to-buffer*. The block of text is copied to the current point in *to-buffer*, and point (in that buffer) is advanced to after the end of the copied text. Is `start`/`end` is `nil`, the entire text in the current buffer is copied over.

See Section 33.19.6 [Sticky Properties], page 935, for other insertion functions that inherit text properties from the nearby text in addition to inserting it. Whitespace inserted by indentation functions also inherits text properties.

33.5 User-Level Insertion Commands

This section describes higher-level commands for inserting text, commands intended primarily for the user but useful also in Lisp programs.

`insert-buffer` *from-buffer-or-name* [Command]

This command inserts the entire accessible contents of *from-buffer-or-name* (which must exist) into the current buffer after point. It leaves the mark after the inserted text. The value is `nil`.

`self-insert-command` *count* **&optional** *char* [Command]

This command inserts the character *char* (the last character typed); it does so *count* times, before point, and returns `nil`. Most printing characters are bound to this command. In routine use, `self-insert-command` is the most frequently called function in Emacs, but programs rarely use it except to install it on a keymap.

In an interactive call, *count* is the numeric prefix argument.

Self-insertion translates the input character through `translation-table-for-input`. See Section 34.9 [Translation of Characters], page 978.

This command calls `auto-fill-function` whenever that is non-`nil` and the character inserted is in the table `auto-fill-chars` (see Section 33.14 [Auto Filling], page 907).

This command performs abbrev expansion if Abbrev mode is enabled and the inserted character does not have word-constituent syntax. (See Chapter 38 [Abbrevs], page 1069, and Section 36.2.1 [Syntax Class Table], page 1026.) It is also responsible for calling `blink-paren-function` when the inserted character has close parenthesis syntax (see Section 41.22 [Blinking], page 1246).

The final thing this command does is to run the hook `post-self-insert-hook`. You could use this to automatically reindent text as it is typed, for example. The functions on this hook can use `last-command-event` (see Section 22.5 [Command Loop Info], page 437) to access the character just inserted.

If any function on this hook needs to act on the region (see Section 32.8 [The Region], page 879), it should make sure Delete Selection mode (see Section "Using Region" in *The GNU Emacs Manual*) doesn't delete the region before `post-self-insert-hook` functions are invoked. The way to do so is to add a function that returns `nil` to `self-insert-uses-region-functions`, a special hook that tells Delete Selection mode it should not delete the region.

Do not try substituting your own definition of `self-insert-command` for the standard one. The editor command loop handles this function specially.

`newline` **&optional** *number-of-newlines interactive* [Command]

This command inserts newlines into the current buffer before point. If *number-of-newlines* is supplied, that many newline characters are inserted. In an interactive call, *number-of-newlines* is the numeric prefix argument.

This command calls `self-insert-command` to insert newlines, which may subsequently break the preceding line by calling `auto-fill-function` (see Section 33.14 [Auto Filling], page 907). Typically what `auto-fill-function` does is insert a newline; thus, the overall result in this case is to insert two newlines at different places: one at point, and another earlier in the line. `newline` does not auto-fill if *number-of-newlines* is non-`nil`.

This command does not run the hook `post-self-insert-hook` unless called interactively or *interactive* is non-`nil`.

This command indents to the left margin if that is not zero. See Section 33.12 [Margins], page 905.

The value returned is `nil`.

`ensure-empty-lines` **&optional** *number-of-empty-lines* [Command]

This command can be used to ensure that you have a specific number of empty lines before point. (An "empty line" is here defined as a line with no characters on it—a line with space characters isn't an empty line.) It defaults to ensuring that there's a single empty line before point.

If point isn't at the beginning of a line, a newline character is inserted first. If there's more empty lines before point than specified, the number of empty lines is reduced. Otherwise it's increased to the specified number.

`overwrite-mode` [Variable]

This variable controls whether overwrite mode is in effect. The value should be `overwrite-mode-textual`, `overwrite-mode-binary`, or `nil`. `overwrite-mode-textual` specifies textual overwrite mode (treats newlines and tabs specially), and `overwrite-mode-binary` specifies binary overwrite mode (treats newlines and tabs like any other characters).

33.6 Deleting Text

Deletion means removing part of the text in a buffer, without saving it in the kill ring (see Section 33.8 [The Kill Ring], page 891). Deleted text can't be yanked, but can be reinserted using the undo mechanism (see Section 33.9 [Undo], page 898). Some deletion functions do save text in the kill ring in some special cases.

All of the deletion functions operate on the current buffer.

`erase-buffer` [Command]

This function deletes the entire text of the current buffer (*not* just the accessible portion), leaving it empty. If the buffer is read-only, it signals a `buffer-read-only` error; if some of the text in it is read-only, it signals a `text-read-only` error. Otherwise, it deletes the text without asking for any confirmation. It returns `nil`.

Normally, deleting a large amount of text from a buffer inhibits further auto-saving of that buffer because it has shrunk. However, `erase-buffer` does not do this, the idea being that the future text is not really related to the former text, and its size should not be compared with that of the former text.

`delete-region` *start end* [Command]

This command deletes the text between positions *start* and *end* in the current buffer, and returns `nil`. If point was inside the deleted region, its value afterward is *start*. Otherwise, point relocates with the surrounding text, as markers do.

`delete-and-extract-region` *start end* [Function]

This function deletes the text between positions *start* and *end* in the current buffer, and returns a string containing the text just deleted.

If point was inside the deleted region, its value afterward is *start*. Otherwise, point relocates with the surrounding text, as markers do.

`delete-char` *count* **&optional** *killp* [Command]

This command deletes *count* characters directly after point, or before point if *count* is negative. If *killp* is non-`nil`, then it saves the deleted characters in the kill ring.

In an interactive call, *count* is the numeric prefix argument, and *killp* is the unprocessed prefix argument. Therefore, if a prefix argument is supplied, the text is saved in the kill ring. If no prefix argument is supplied, then one character is deleted, but not saved in the kill ring.

The value returned is always `nil`.

`delete-backward-char` *count* **&optional** *killp* [Command]

This command deletes *count* characters directly before point, or after point if *count* is negative. If *killp* is non-`nil`, then it saves the deleted characters in the kill ring.

In an interactive call, *count* is the numeric prefix argument, and *killp* is the unprocessed prefix argument. Therefore, if a prefix argument is supplied, the text is saved in the kill ring. If no prefix argument is supplied, then one character is deleted, but not saved in the kill ring.

The value returned is always `nil`.

`backward-delete-char-untabify` *count* **&optional** *killp* [Command]
This command deletes *count* characters backward, changing tabs into spaces. When the next character to be deleted is a tab, it is first replaced with the proper number of spaces to preserve alignment and then one of those spaces is deleted instead of the tab. If *killp* is non-`nil`, then the command saves the deleted characters in the kill ring.

Conversion of tabs to spaces happens only if *count* is positive. If it is negative, exactly −*count* characters after point are deleted.

In an interactive call, *count* is the numeric prefix argument, and *killp* is the unprocessed prefix argument. Therefore, if a prefix argument is supplied, the text is saved in the kill ring. If no prefix argument is supplied, then one character is deleted, but not saved in the kill ring.

The value returned is always `nil`.

`backward-delete-char-untabify-method` [User Option]
This option specifies how `backward-delete-char-untabify` should deal with whitespace. Possible values include `untabify`, the default, meaning convert a tab to many spaces and delete one; `hungry`, meaning delete all tabs and spaces before point with one command; `all` meaning delete all tabs, spaces and newlines before point, and `nil`, meaning do nothing special for whitespace characters.

33.7 User-Level Deletion Commands

This section describes higher-level commands for deleting text, commands intended primarily for the user but useful also in Lisp programs.

`delete-horizontal-space` **&optional** *backward-only* [Command]
This function deletes all spaces and tabs around point. It returns `nil`.

If *backward-only* is non-`nil`, the function deletes spaces and tabs before point, but not after point.

In the following examples, we call `delete-horizontal-space` four times, once on each line, with point between the second and third characters on the line each time.

```
---------- Buffer: foo ----------
I ⋆thought
I ⋆     thought
We⋆ thought
Yo⋆u thought
---------- Buffer: foo ----------
```

```
(delete-horizontal-space)   ; Four times.
     ⇒ nil

---------- Buffer: foo ----------
Ithought
Ithought
Wethought
You thought
---------- Buffer: foo ----------
```

`delete-indentation` **&optional** *join-following-p beg end* [Command]

This function joins the line point is on to the previous line, deleting any whitespace at the join and in some cases replacing it with one space. If *join-following-p* is non-`nil`, `delete-indentation` joins this line to the following line instead. Otherwise, if *beg* and *end* are non-`nil`, this function joins all lines in the region they define.

In an interactive call, *join-following-p* is the prefix argument, and *beg* and *end* are, respectively, the start and end of the region if it is active, else `nil`. The function returns `nil`.

If there is a fill prefix, and the second of the lines being joined starts with the prefix, then `delete-indentation` deletes the fill prefix before joining the lines. See Section 33.12 [Margins], page 905.

In the example below, point is located on the line starting '`events`', and it makes no difference if there are trailing spaces in the preceding line.

```
---------- Buffer: foo ----------
When in the course of human
⋆    events, it becomes necessary
---------- Buffer: foo ----------

(delete-indentation)
     ⇒ nil

---------- Buffer: foo ----------
When in the course of human⋆ events, it becomes necessary
---------- Buffer: foo ----------
```

After the lines are joined, the function `fixup-whitespace` is responsible for deciding whether to leave a space at the junction.

`fixup-whitespace` [Command]

This function replaces all the horizontal whitespace surrounding point with either one space or no space, according to the context. It returns `nil`.

At the beginning or end of a line, the appropriate amount of space is none. Before a character with close parenthesis syntax, or after a character with open parenthesis or expression-prefix syntax, no space is also appropriate. Otherwise, one space is appropriate. See Section 36.2.1 [Syntax Class Table], page 1026.

In the example below, `fixup-whitespace` is called the first time with point before the word '`spaces`' in the first line. For the second invocation, point is directly after the '`(`'.

```
---------- Buffer: foo ----------
This has too many     ⋆spaces
This has too many spaces at the start of (⋆   this list)
---------- Buffer: foo ----------

(fixup-whitespace)
     ⇒ nil
(fixup-whitespace)
     ⇒ nil

---------- Buffer: foo ----------
This has too many spaces
This has too many spaces at the start of (this list)
---------- Buffer: foo ----------
```

`just-one-space` **&optional** *n* [Command]
: This command replaces any spaces and tabs around point with a single space, or *n* spaces if *n* is specified. It returns `nil`.

`delete-blank-lines` [Command]
: This function deletes blank lines surrounding point. If point is on a blank line with one or more blank lines before or after it, then all but one of them are deleted. If point is on an isolated blank line, then it is deleted. If point is on a nonblank line, the command deletes all blank lines immediately following it.

  A blank line is defined as a line containing only tabs and spaces.

  `delete-blank-lines` returns `nil`.

`delete-trailing-whitespace` **&optional** *start end* [Command]
: Delete trailing whitespace in the region defined by *start* and *end*.

  This command deletes whitespace characters after the last non-whitespace character in each line in the region.

  If this command acts on the entire buffer (i.e., if called interactively with the mark inactive, or called from Lisp with *end* `nil`), it also deletes all trailing lines at the end of the buffer if the variable `delete-trailing-lines` is non-`nil`.

33.8 The Kill Ring

Kill functions delete text like the deletion functions, but save it so that the user can reinsert it by *yanking*. Most of these functions have '`kill-`' in their name. By contrast, the functions whose names start with '`delete-`' normally do not save text for yanking (though they can still be undone); these are deletion functions.

Most of the kill commands are primarily for interactive use, and are not described here. What we do describe are the functions provided for use in writing such commands. You can use these functions to write commands for killing text. When you need to delete text for internal purposes within a Lisp function, you should normally use deletion functions, so as not to disturb the kill ring contents. See Section 33.6 [Deletion], page 888.

Killed text is saved for later yanking in the *kill ring*. This is a list that holds a number of recent kills, not just the last text kill. We call this a "ring" because yanking treats it as having elements in a cyclic order. The list is kept in the variable `kill-ring`, and can be

operated on with the usual functions for lists; there are also specialized functions, described in this section, that treat it as a ring.

Some people think this use of the word "kill" is unfortunate, since it refers to operations that specifically *do not* destroy the entities killed. This is in sharp contrast to ordinary life, in which death is permanent and killed entities do not come back to life. Therefore, other metaphors have been proposed. For example, the term "cut ring" makes sense to people who, in pre-computer days, used scissors and paste to cut up and rearrange manuscripts. However, it would be difficult to change the terminology now.

33.8.1 Kill Ring Concepts

The kill ring records killed text as strings in a list, most recent first. A short kill ring, for example, might look like this:

```
("some text" "a different piece of text" "even older text")
```

When the list reaches `kill-ring-max` entries in length, adding a new entry automatically deletes the last entry.

When kill commands are interwoven with other commands, each kill command makes a new entry in the kill ring. Multiple kill commands in succession build up a single kill ring entry, which would be yanked as a unit; the second and subsequent consecutive kill commands add text to the entry made by the first one.

For yanking, one entry in the kill ring is designated the front of the ring. Some yank commands rotate the ring by designating a different element as the front. But this virtual rotation doesn't change the list itself—the most recent entry always comes first in the list.

33.8.2 Functions for Killing

`kill-region` is the usual subroutine for killing text. Any command that calls this function is a kill command (and should probably have '`kill`' in its name). `kill-region` puts the newly killed text in a new element at the beginning of the kill ring or adds it to the most recent element. It determines automatically (using `last-command`) whether the previous command was a kill command, and if so appends the killed text to the most recent entry.

The commands described below can filter the killed text before they save it in the kill ring. They call `filter-buffer-substring` (see Section 33.2 [Buffer Contents], page 881) to perform the filtering. By default, there's no filtering, but major and minor modes and hook functions can set up filtering, so that text saved in the kill ring is different from what was in the buffer.

`kill-region` *start end* **&optional** *region* [Command]

This function kills the stretch of text between *start* and *end*; but if the optional argument *region* is non-`nil`, it ignores *start* and *end*, and kills the text in the current region instead. The text is deleted but saved in the kill ring, along with its text properties. The value is always `nil`.

In an interactive call, *start* and *end* are point and the mark, and *region* is always non-`nil`, so the command always kills the text in the current region.

If the buffer or text is read-only, `kill-region` modifies the kill ring just the same, then signals an error without modifying the buffer. This is convenient because it lets the user use a series of kill commands to copy text from a read-only buffer into the kill ring.

`kill-read-only-ok` [User Option]
: If this option is non-`nil`, `kill-region` does not signal an error if the buffer or text is read-only. Instead, it simply returns, updating the kill ring but not changing the buffer.

`copy-region-as-kill` *start end* **&optional** *region* [Command]
: This function saves the stretch of text between *start* and *end* on the kill ring (including text properties), but does not delete the text from the buffer. However, if the optional argument *region* is non-`nil`, the function ignores *start* and *end*, and saves the current region instead. It always returns `nil`.

  In an interactive call, *start* and *end* are point and the mark, and *region* is always non-`nil`, so the command always saves the text in the current region.

  The command does not set `this-command` to `kill-region`, so a subsequent kill command does not append to the same kill ring entry.

33.8.3 Yanking

Yanking means inserting text from the kill ring, but it does not insert the text blindly. The `yank` command, and related commands, use `insert-for-yank` to perform special processing on the text before it is inserted.

`insert-for-yank` *string* [Function]
: This function works like `insert`, except that it processes the text in *string* according to the `yank-handler` text property, as well as the variables `yank-handled-properties` and `yank-excluded-properties` (see below), before inserting the result into the current buffer.

  string will be run through `yank-transform-functions` (see below) before inserting.

`insert-buffer-substring-as-yank` *buf* **&optional** *start end* [Function]
: This function resembles `insert-buffer-substring`, except that it processes the text according to `yank-handled-properties` and `yank-excluded-properties`. (It does not handle the `yank-handler` property, which does not normally occur in buffer text anyway.)

If you put a `yank-handler` text property on all or part of a string, that alters how `insert-for-yank` inserts the string. If different parts of the string have different `yank-handler` values (comparison being done with `eq`), each substring is handled separately. The property value must be a list of one to four elements, with the following format (where elements after the first may be omitted):

```
(function param noexclude undo)
```

Here is what the elements do:

function
: When *function* is non-`nil`, it is called instead of `insert` to insert the string, with one argument—the string to insert.

param
: If *param* is present and non-`nil`, it replaces *string* (or the substring of *string* being processed) as the object passed to *function* (or `insert`). For example, if *function* is `yank-rectangle`, *param* should be a list of strings to insert as a rectangle.

noexclude If *noexclude* is present and non-`nil`, that disables the normal action of `yank-handled-properties` and `yank-excluded-properties` on the inserted string.

undo If *undo* is present and non-`nil`, it is a function that will be called by `yank-pop` to undo the insertion of the current object. It is called with two arguments, the start and end of the current region. *function* can set `yank-undo-function` to override the *undo* value.

`yank-handled-properties` [User Option]
This variable specifies special text property handling conditions for yanked text. It takes effect after the text has been inserted (either normally, or via the `yank-handler` property), and prior to `yank-excluded-properties` taking effect.

The value should be an alist of elements `(`*`prop`*` . `*`fun`*`)`. Each alist element is handled in order. The inserted text is scanned for stretches of text having text properties `eq` to *prop*; for each such stretch, *fun* is called with three arguments: the value of the property, and the start and end positions of the text.

`yank-excluded-properties` [User Option]
The value of this variable is the list of properties to remove from inserted text. Its default value contains properties that might lead to annoying results, such as causing the text to respond to the mouse or specifying key bindings. It takes effect after `yank-handled-properties`.

`yank-transform-functions` [Variable]
This variable is a list of functions. Each function is called (in order) with the string to be yanked as the argument, and should return a (possibly transformed) string. This variable can be set globally, but can also be used to create new commands that are variations on `yank`. For instance, to create a command that works like `yank`, but cleans up whitespace before inserting, you could say something like:

```
(defun yank-with-clean-whitespace ()
  (interactive)
  (let ((yank-transform-functions
 '(string-clean-whitespace)))
    (call-interactively #'yank)))
```

33.8.4 Functions for Yanking

This section describes higher-level commands for yanking, which are intended primarily for the user but useful also in Lisp programs. Both `yank` and `yank-pop` honor the `yank-excluded-properties` variable and `yank-handler` text property (see Section 33.8.3 [Yanking], page 893).

`yank` **&optional** *arg* [Command]
This command inserts before point the text at the front of the kill ring. It sets the mark at the beginning of that text, using `push-mark` (see Section 32.7 [The Mark], page 875), and puts point at the end.

If *arg* is a non-`nil` list (which occurs interactively when the user types `C-u` with no digits), then `yank` inserts the text as described above, but puts point before the yanked text and sets the mark after it.

If *arg* is a number, then `yank` inserts the *arg*th most recently killed text—the *arg*th element of the kill ring list, counted cyclically from the front, which is considered the first element for this purpose.

`yank` does not alter the contents of the kill ring, unless it used text provided by another program, in which case it pushes that text onto the kill ring. However if *arg* is an integer different from one, it rotates the kill ring to place the yanked string at the front.

`yank` returns `nil`.

`yank-pop` **&optional** *arg* [Command]

When invoked immediately after a `yank` or another `yank-pop`, this command replaces the just-yanked entry from the kill ring with a different entry from the kill ring. When this command is invoked like that, the region contains text that was just inserted by another yank command. `yank-pop` deletes that text and inserts in its place a different piece of killed text. It does not add the deleted text to the kill ring, since it is already in the kill ring somewhere. It does however rotate the kill ring to place the newly yanked string at the front.

If *arg* is `nil`, then the replacement text is the previous element of the kill ring. If *arg* is numeric, the replacement is the *arg*th previous kill. If *arg* is negative, a more recent kill is the replacement.

The sequence of kills in the kill ring wraps around, so if `yank-pop` is invoked repeatedly and reaches the oldest kill, the one that comes after it is the newest one, and the one before the newest one is the oldest one.

This command can also be invoked after a command that is not a yank command. In that case, it prompts in the minibuffer for a kill-ring entry, with completion, and uses the kill ring elements as the minibuffer history (see Section 21.4 [Minibuffer History], page 393). This allows the user to interactively select one of the previous kills recorded in the kill ring.

The return value is always `nil`.

`yank-undo-function` [Variable]

If this variable is non-`nil`, the function `yank-pop` uses its value instead of `delete-region` to delete the text inserted by the previous `yank` or `yank-pop` command. The value must be a function of two arguments, the start and end of the current region.

The function `insert-for-yank` automatically sets this variable according to the *undo* element of the `yank-handler` text property, if there is one.

33.8.5 Low-Level Kill Ring

These functions and variables provide access to the kill ring at a lower level, but are still convenient for use in Lisp programs, because they take care of interaction with window system selections (see Section 30.20 [Window System Selections], page 842).

`current-kill` *n* **&optional** *do-not-move* [Function]

The function `current-kill` rotates the yanking pointer, which designates the front of the kill ring, by *n* places (from newer kills to older ones), and returns the text at that place in the ring.

If the optional second argument *do-not-move* is non-`nil`, then `current-kill` doesn't alter the yanking pointer; it just returns the *n*th kill, counting from the current yanking pointer.

If *n* is zero, indicating a request for the latest kill, `current-kill` calls the value of `interprogram-paste-function` (documented below) before consulting the kill ring. If that value is a function and calling it returns a string or a list of several strings, `current-kill` pushes the strings onto the kill ring and returns the first string. It also sets the yanking pointer to point to the kill-ring entry of the first string returned by `interprogram-paste-function`, regardless of the value of *do-not-move*. Otherwise, `current-kill` does not treat a zero value for *n* specially: it returns the entry pointed at by the yanking pointer and does not move the yanking pointer.

`kill-new` *string* **&optional** *replace* [Function]

This function pushes the text *string* onto the kill ring and makes the yanking pointer point to it. It discards the oldest entry if appropriate. It also invokes the values of `interprogram-paste-function` (subject to the user option `save-interprogram-paste-before-kill`) and `interprogram-cut-function` (see below).

If *replace* is non-`nil`, then `kill-new` replaces the first element of the kill ring with *string*, rather than pushing *string* onto the kill ring.

`kill-append` *string before-p* [Function]

This function appends the text *string* to the first entry in the kill ring and makes the yanking pointer point to the combined entry. Normally *string* goes at the end of the entry, but if *before-p* is non-`nil`, it goes at the beginning. This function calls `kill-new` as a subroutine, thus causing the values of `interprogram-cut-function` and possibly `interprogram-paste-function` (see below) to be invoked by extension.

`interprogram-paste-function` [Variable]

This variable provides a way of transferring killed text from other programs, when you are using a window system. Its value should be `nil` or a function of no arguments.

If the value is a function, `current-kill` calls it to get the most recent kill. If the function returns a non-`nil` value, then that value is used as the most recent kill. If it returns `nil`, then the front of the kill ring is used.

To facilitate support for window systems that support multiple selections, this function may also return a list of strings. In that case, the first string is used as the most recent kill, and all the other strings are pushed onto the kill ring, for easy access by `yank-pop`.

The normal use of this function is to get the window system's clipboard as the most recent kill, even if the selection belongs to another application. See Section 30.20 [Window System Selections], page 842. However, if the clipboard contents come from the current Emacs session, this function should return `nil`.

`interprogram-cut-function` [Variable]

This variable provides a way of communicating killed text to other programs, when you are using a window system. Its value should be `nil` or a function of one required argument.

If the value is a function, `kill-new` and `kill-append` call it with the new first element of the kill ring as the argument.

The normal use of this function is to put newly killed text in the window system's clipboard. See Section 30.20 [Window System Selections], page 842.

33.8.6 Internals of the Kill Ring

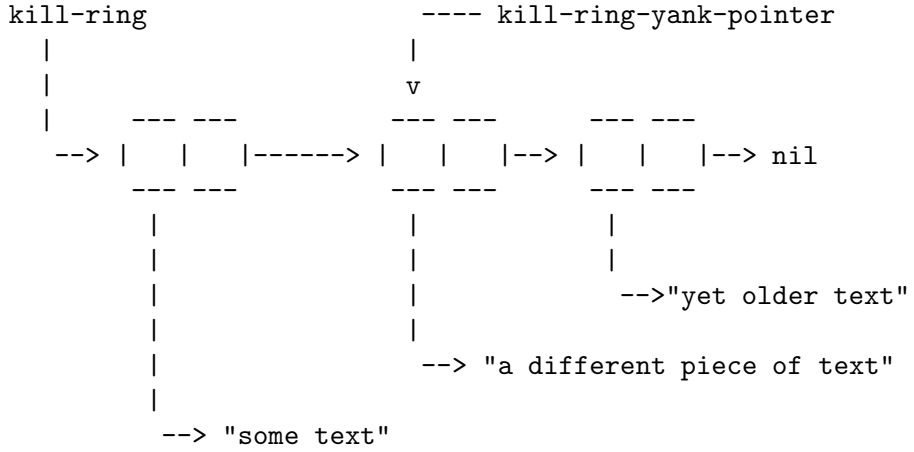
The variable `kill-ring` holds the kill ring contents, in the form of a list of strings. The most recent kill is always at the front of the list.

The `kill-ring-yank-pointer` variable points to a link in the kill ring list, whose CAR is the text to yank next. We say it identifies the front of the ring. Moving `kill-ring-yank-pointer` to a different link is called *rotating the kill ring*. We call the kill ring a "ring" because the functions that move the yank pointer wrap around from the end of the list to the beginning, or vice-versa. Rotation of the kill ring is virtual; it does not change the value of `kill-ring`.

Both `kill-ring` and `kill-ring-yank-pointer` are Lisp variables whose values are normally lists. The word "pointer" in the name of the `kill-ring-yank-pointer` indicates that the variable's purpose is to identify one element of the list for use by the next yank command.

The value of `kill-ring-yank-pointer` is always `eq` to one of the links in the kill ring list. The element it identifies is the CAR of that link. Kill commands, which change the kill ring, also set this variable to the value of `kill-ring`. The effect is to rotate the ring so that the newly killed text is at the front.

Here is a diagram that shows the variable `kill-ring-yank-pointer` pointing to the second entry in the kill ring `("some text" "a different piece of text" "yet older text")`.



This state of affairs might occur after `C-y` (`yank`) immediately followed by `M-y` (`yank-pop`).

`kill-ring` [Variable]

This variable holds the list of killed text sequences, most recently killed first.

`kill-ring-yank-pointer` [Variable]

This variable's value indicates which element of the kill ring is at the front of the ring for yanking. More precisely, the value is a tail of the value of `kill-ring`, and its CAR is the kill string that `C-y` should yank.

`kill-ring-max` [User Option]
: The value of this variable is the maximum length to which the kill ring can grow, before elements are thrown away at the end. The default value for `kill-ring-max` is 120.

33.9 Undo

Most buffers have an *undo list*, which records all changes made to the buffer's text so that they can be undone. (The buffers that don't have one are usually special-purpose buffers for which Emacs assumes that undoing is not useful. In particular, any buffer whose name begins with a space has its undo recording off by default; see Section 28.3 [Buffer Names], page 676.) All the primitives that modify the text in the buffer automatically add elements to the front of the undo list, which is in the variable `buffer-undo-list`.

`buffer-undo-list` [Variable]
: This buffer-local variable's value is the undo list of the current buffer. A value of `t` disables the recording of undo information.

Here are the kinds of elements an undo list can have:

position
: This kind of element records a previous value of point; undoing this element moves point to *position*. Ordinary cursor motion does not make any sort of undo record, but deletion operations use these entries to record where point was before the command.

(*beg* . *end*)
: This kind of element indicates how to delete text that was inserted. Upon insertion, the text occupied the range *beg*–*end* in the buffer.

(*text* . *position*)
: This kind of element indicates how to reinsert text that was deleted. The deleted text itself is the string *text*. The place to reinsert it is `(abs` *position*`)`. If *position* is positive, point was at the beginning of the deleted text, otherwise it was at the end. Zero or more (*marker* . *adjustment*) elements follow immediately after this element.

(`t` . *time-flag*)
: This kind of element indicates that an unmodified buffer became modified. A *time-flag* that is a non-integer Lisp timestamp represents the visited file's modification time as of when it was previously visited or saved, using the same format as `current-time`; see Section 42.5 [Time of Day], page 1275. A *time-flag* of 0 means the buffer does not correspond to any file; −1 means the visited file previously did not exist. `primitive-undo` uses these values to determine whether to mark the buffer as unmodified once again; it does so only if the file's status matches that of *time-flag*.

(`nil` *property value beg* . *end*)
: This kind of element records a change in a text property. Here's how you might undo the change:

```
(put-text-property beg end property value)
```

`(`*marker* `.` *adjustment*`)`
: This kind of element records the fact that the marker *marker* was relocated due to deletion of surrounding text, and that it moved *adjustment* character positions. If the marker's location is consistent with the (*text* . *position*) element preceding it in the undo list, then undoing this element moves *marker* − *adjustment* characters.

`(apply` *funname* `.` *args*`)`
: This is an extensible undo item, which is undone by calling *funname* with arguments *args*.

`(apply` *delta beg end funname* `.` *args*`)`
: This is an extensible undo item, which records a change limited to the range *beg* to *end*, which increased the size of the buffer by *delta* characters. It is undone by calling *funname* with arguments *args*.

  This kind of element enables undo limited to a region to determine whether the element pertains to that region.

`nil`
: This element is a boundary. The elements between two boundaries are called a *change group*; normally, each change group corresponds to one keyboard command, and undo commands normally undo an entire group as a unit.

`undo-boundary` [Function]

This function places a boundary element in the undo list. The undo command stops at such a boundary, and successive undo commands undo to earlier and earlier boundaries. This function returns `nil`.

Calling this function explicitly is useful for splitting the effects of a command into more than one unit. For example, `query-replace` calls `undo-boundary` after each replacement, so that the user can undo individual replacements one by one.

Mostly, however, this function is called automatically at an appropriate time.

`undo-auto-amalgamate` [Function]

The editor command loop automatically calls `undo-boundary` just before executing each key sequence, so that each undo normally undoes the effects of one command. A few exceptional commands are *amalgamating*: these commands generally cause small changes to buffers, so with these a boundary is inserted only every 20th command, allowing the changes to be undone as a group. By default, the commands `self-insert-command`, which produces self-inserting input characters (see Section 33.5 [Commands for Insertion], page 886), and `delete-char`, which deletes characters (see Section 33.6 [Deletion], page 888), are amalgamating. Where a command affects the contents of several buffers, as may happen, for example, when a function on the `post-command-hook` affects a buffer other than the `current-buffer`, then `undo-boundary` will be called in each of the affected buffers.

This function can be called before an amalgamating command. It removes the previous `undo-boundary` if a series of such calls have been made.

The maximum number of changes that can be amalgamated is controlled by the `amalgamating-undo-limit` variable. If this variable is 1, no changes are amalgamated.

A Lisp program can amalgamate a series of changes into a single change group by calling `undo-amalgamate-change-group` (see Section 33.33 [Atomic Changes], page 962). Note that `amalgamating-undo-limit` has no effect on the groups produced by that function.

`undo-auto-current-boundary-timer` [Variable]
: Some buffers, such as process buffers, can change even when no commands are executing. In these cases, `undo-boundary` is normally called periodically by the timer in this variable. Setting this variable to non-`nil` prevents this behavior.

`undo-in-progress` [Variable]
: This variable is normally `nil`, but the undo commands bind it to `t`. This is so that various kinds of change hooks can tell when they're being called for the sake of undoing.

`primitive-undo` *count list* [Function]
: This is the basic function for undoing elements of an undo list. It undoes the first *count* elements of *list*, returning the rest of *list*.

  `primitive-undo` adds elements to the buffer's undo list when it changes the buffer. Undo commands avoid confusion by saving the undo list value at the beginning of a sequence of undo operations. Then the undo operations use and update the saved value. The new elements added by undoing are not part of this saved value, so they don't interfere with continuing to undo.

  This function does not bind `undo-in-progress`.

`with-undo-amalgamate` *body...* [Macro]
: This macro removes all the undo boundaries inserted during the execution of *body* so that it can be undone as a single step.

Some commands leave the region active after execution in such a way that it interferes with selective undo of that command. To make `undo` ignore the active region when invoked immediately after such a command, set the property `undo-inhibit-region` of the command's function symbol to a non-`nil` value. See Section 9.4.2 [Standard Properties], page 139.

33.10 Maintaining Undo Lists

This section describes how to enable and disable undo information for a given buffer. It also explains how the undo list is truncated automatically so it doesn't get too big.

Recording of undo information in a newly created buffer is normally enabled to start with; but if the buffer name starts with a space, the undo recording is initially disabled. You can explicitly enable or disable undo recording with the following two functions, or by setting `buffer-undo-list` yourself.

`buffer-enable-undo` **&optional** *buffer-or-name* [Command]
: This command enables recording undo information for buffer *buffer-or-name*, so that subsequent changes can be undone. If no argument is supplied, then the current buffer is used. This function does nothing if undo recording is already enabled in the buffer. It returns `nil`.

  In an interactive call, *buffer-or-name* is the current buffer. You cannot specify any other buffer.

`buffer-disable-undo` **&optional** *buffer-or-name* [Command]
: This function discards the undo list of *buffer-or-name*, and disables further recording of undo information. As a result, it is no longer possible to undo either previous changes or any subsequent changes. If the undo list of *buffer-or-name* is already disabled, this function has no effect.

  In an interactive call, BUFFER-OR-NAME is the current buffer. You cannot specify any other buffer. This function returns `nil`.

As editing continues, undo lists get longer and longer. To prevent them from using up all available memory space, garbage collection trims them back to size limits you can set. (For this purpose, the size of an undo list measures the cons cells that make up the list, plus the strings of deleted text.) Three variables control the range of acceptable sizes: `undo-limit`, `undo-strong-limit` and `undo-outer-limit`. In these variables, size is counted as the number of bytes occupied, which includes both saved text and other data.

`undo-limit` [User Option]
: This is the soft limit for the acceptable size of an undo list. The change group at which this size is exceeded is the last one kept.

`undo-strong-limit` [User Option]
: This is the upper limit for the acceptable size of an undo list. The change group at which this size is exceeded is discarded itself (along with all older change groups). There is one exception: the very latest change group is only discarded if it exceeds `undo-outer-limit`.

`undo-outer-limit` [User Option]
: If at garbage collection time the undo info for the current command exceeds this limit, Emacs discards the info and displays a warning. This is a last ditch limit to prevent memory overflow.

`undo-ask-before-discard` [User Option]
: If this variable is non-`nil`, when the undo info exceeds `undo-outer-limit`, Emacs asks in the echo area whether to discard the info. The default value is `nil`, which means to discard it automatically.

  This option is mainly intended for debugging. Garbage collection is inhibited while the question is asked, which means that Emacs might leak memory if the user waits too long before answering the question.

33.11 Filling

Filling means adjusting the lengths of lines (by moving the line breaks) so that they are nearly (but no greater than) a specified maximum width. Additionally, lines can be *justified*, which means inserting spaces to make the left and/or right margins line up precisely. The width is controlled by the variable `fill-column`. For ease of reading, lines should be no longer than 70 or so columns.

You can use Auto Fill mode (see Section 33.14 [Auto Filling], page 907) to fill text automatically as you insert it, but changes to existing text may leave it improperly filled. Then you must fill the text explicitly.

Most of the commands in this section return values that are not meaningful. All the functions that do filling take note of the current left margin, current right margin, and current justification style (see Section 33.12 [Margins], page 905). If the current justification style is `none`, the filling functions don't actually do anything.

Several of the filling functions have an argument *justify*. If it is non-`nil`, that requests some kind of justification. It can be `left`, `right`, `full`, or `center`, to request a specific style of justification. If it is `t`, that means to use the current justification style for this part of the text (see `current-justification`, below). Any other value is treated as `full`.

When you call the filling functions interactively, using a prefix argument implies the value `full` for *justify*.

`fill-paragraph` **&optional** *justify region* [Command]
: This command fills the paragraph at or after point. If *justify* is non-`nil`, each line is justified as well. It uses the ordinary paragraph motion commands to find paragraph boundaries. See Section "Paragraphs" in *The GNU Emacs Manual*.

  When *region* is non-`nil`, then if Transient Mark mode is enabled and the mark is active, this command calls `fill-region` to fill all the paragraphs in the region, instead of filling only the current paragraph. When this command is called interactively, *region* is `t`.

`fill-region` *start end* **&optional** *justify nosqueeze to-eop* [Command]
: This command fills each of the paragraphs in the region from *start* to *end*. It justifies as well if *justify* is non-`nil`.

  If *nosqueeze* is non-`nil`, that means to leave whitespace other than line breaks untouched. If *to-eop* is non-`nil`, that means to keep filling to the end of the paragraph—or the next hard newline, if `use-hard-newlines` is enabled (see below).

  The variable `paragraph-separate` controls how to distinguish paragraphs. See Section 35.8 [Standard Regexps], page 1023.

`pixel-fill-region` *start end pixel-width* [Function]
: Most Emacs buffers use monospaced text, so all the filling functions (like `fill-region`) work based on the number of characters and `char-width`. However, Emacs can render other types of things, like text that contains images and using proportional fonts, and the `pixel-fill-region` exists to handle that. It fills the region of text between *start* and *end* at pixel granularity, so text using variable-pitch fonts or several different fonts looks filled regardless of different character sizes. The argument *pixel-width* specifies the maximum pixel width a line is allowed to have after filling; it is the pixel-resolution equivalent of the `fill-column` in `fill-region`. For instance, this Lisp snippet will insert text using a proportional font, and then fill this to be no wider than 300 pixels:

  ```
  (insert (propertize
   "This is a sentence that's ends here."
   'face 'variable-pitch))
  (pixel-fill-region (point) (point-max) 300)
  ```

  If *start* isn't at the start of a line, the horizontal position of *start*, converted to pixel units, will be used as the indentation prefix on subsequent lines.

The `pixel-fill-width` helper function can be used to compute the pixel width to use. If given no arguments, it'll return a value slightly less than the width of the current window. The first optional value, *columns*, specifies the number of columns using the standard, monospaced fonts, e.g. `fill-column`. The second optional value is the window to use. You'd typically use it like this:

```
(pixel-fill-region
  start end (pixel-fill-width fill-column))
```

`fill-individual-paragraphs` *start end* **&optional** *justify citation-regexp* [Command]

This command fills each paragraph in the region according to its individual fill prefix. Thus, if the lines of a paragraph were indented with spaces, the filled paragraph will remain indented in the same fashion.

The first two arguments, *start* and *end*, are the beginning and end of the region to be filled. The third and fourth arguments, *justify* and *citation-regexp*, are optional. If *justify* is non-`nil`, the paragraphs are justified as well as filled. If *citation-regexp* is non-`nil`, it means the function is operating on a mail message and therefore should not fill the header lines. If *citation-regexp* is a string, it is used as a regular expression; if it matches the beginning of a line, that line is treated as a citation marker.

Ordinarily, `fill-individual-paragraphs` regards each change in indentation as starting a new paragraph. If `fill-individual-varying-indent` is non-`nil`, then only separator lines separate paragraphs. That mode can handle indented paragraphs with additional indentation on the first line.

`fill-individual-varying-indent` [User Option]

This variable alters the action of `fill-individual-paragraphs` as described above.

`fill-region-as-paragraph` *start end* **&optional** *justify nosqueeze squeeze-after* [Command]

This command considers a region of text as a single paragraph and fills it. If the region was made up of many paragraphs, the blank lines between paragraphs are removed. This function justifies as well as filling when *justify* is non-`nil`.

If *nosqueeze* is non-`nil`, that means to leave whitespace other than line breaks untouched. If *squeeze-after* is non-`nil`, it specifies a position in the region, and means that whitespace other than line breaks should be left untouched before that position.

In Adaptive Fill mode, this command calls `fill-context-prefix` to choose a fill prefix by default. See Section 33.13 [Adaptive Fill], page 906.

`justify-current-line` **&optional** *how eop nosqueeze* [Command]

This command inserts spaces between the words of the current line so that the line ends exactly at `fill-column`. It returns `nil`.

The argument *how*, if non-`nil` specifies explicitly the style of justification. It can be `left`, `right`, `full`, `center`, or `none`. If it is `t`, that means to follow specified justification style (see `current-justification`, below). `nil` means to do full justification.

If *eop* is non-`nil`, that means do only left-justification if `current-justification` specifies full justification. This is used for the last line of a paragraph; even if the paragraph as a whole is fully justified, the last line should not be.

If *nosqueeze* is non-`nil`, that means do not change interior whitespace.

`default-justification` [User Option]
This variable's value specifies the style of justification to use for text that doesn't specify a style with a text property. The possible values are `left`, `right`, `full`, `center`, or `none`. The default value is `left`.

`current-justification` [Function]
This function returns the proper justification style to use for filling the text around point.

This returns the value of the `justification` text property at point, or the variable `default-justification` if there is no such text property. However, it returns `nil` rather than `none` to mean "don't justify".

`sentence-end-double-space` [User Option]
If this variable is non-`nil`, a period followed by just one space does not count as the end of a sentence, and the filling functions avoid breaking the line at such a place.

`sentence-end-without-period` [User Option]
If this variable is non-`nil`, a sentence can end without a period. This is used for languages like Thai, where sentences end with a double space but without a period.

`sentence-end-without-space` [User Option]
If this variable is non-`nil`, it should be a string of characters that can end a sentence without following spaces.

`fill-separate-heterogeneous-words-with-space` [User Option]
If this variable is non-`nil`, two words of different kind (e.g., English and CJK) will be separated with a space when concatenating one that is in the end of a line and the other that is in the beginning of the next line for filling.

`fill-paragraph-function` [Variable]
This variable provides a way to override the filling of paragraphs. If its value is non-`nil`, `fill-paragraph` calls this function to do the work. If the function returns a non-`nil` value, `fill-paragraph` assumes the job is done, and immediately returns that value.

The usual use of this feature is to fill comments in programming language modes. If the function needs to fill a paragraph in the usual way, it can do so as follows:

```
(let ((fill-paragraph-function nil))
  (fill-paragraph arg))
```

`fill-forward-paragraph-function` [Variable]
This variable provides a way to override how the filling functions, such as `fill-region` and `fill-paragraph`, move forward to the next paragraph. Its value should be a function, which is called with a single argument *n*, the number of paragraphs to move, and should return the difference between *n* and the number of paragraphs actually moved. The default value of this variable is `forward-paragraph`. See Section "Paragraphs" in *The GNU Emacs Manual*.

`use-hard-newlines` [Variable]
: If this variable is non-`nil`, the filling functions do not delete newlines that have the `hard` text property. These hard newlines act as paragraph separators. See Section "Hard and Soft Newlines" in *The GNU Emacs Manual*.

33.12 Margins for Filling

`fill-prefix` [User Option]
: This buffer-local variable, if non-`nil`, specifies a string of text that appears at the beginning of normal text lines and should be disregarded when filling them. Any line that fails to start with the fill prefix is considered the start of a paragraph; so is any line that starts with the fill prefix followed by additional whitespace. Lines that start with the fill prefix but no additional whitespace are ordinary text lines that can be filled together. The resulting filled lines also start with the fill prefix.

  The fill prefix follows the left margin whitespace, if any.

`fill-column` [User Option]
: This buffer-local variable specifies the maximum width of filled lines. Its value should be an integer, which is a number of columns. All the filling, justification, and centering commands are affected by this variable, including Auto Fill mode (see Section 33.14 [Auto Filling], page 907).

  As a practical matter, if you are writing text for other people to read, you should set `fill-column` to no more than 70. Otherwise the line will be too long for people to read comfortably, and this can make the text seem clumsy.

  The default value for `fill-column` is 70. To disable Auto Fill mode in a specific mode, you could say something like:

```
(add-hook 'foo-mode-hook (lambda () (auto-fill-mode -1))
```

`set-left-margin` *from to margin* [Command]
: This sets the `left-margin` property on the text from *from* to *to* to the value *margin*. If Auto Fill mode is enabled, this command also refills the region to fit the new margin.

`set-right-margin` *from to margin* [Command]
: This sets the `right-margin` property on the text from *from* to *to* to the value *margin*. If Auto Fill mode is enabled, this command also refills the region to fit the new margin.

`current-left-margin` [Function]
: This function returns the proper left margin value to use for filling the text around point. The value is the sum of the `left-margin` property of the character at the start of the current line (or zero if none), and the value of the variable `left-margin`.

`current-fill-column` [Function]
: This function returns the proper fill column value to use for filling the text around point. The value is the value of the `fill-column` variable, minus the value of the `right-margin` property of the character after point.

`move-to-left-margin` **&optional** *n force* [Command]
: This function moves point to the left margin of the current line. The column moved to is determined by calling the function `current-left-margin`. If the argument *n* is non-`nil`, `move-to-left-margin` moves forward $n-1$ lines first.

  If *force* is non-`nil`, that says to fix the line's indentation if that doesn't match the left margin value.

`delete-to-left-margin` **&optional** *from to* [Function]
: This function removes left margin indentation from the text between *from* and *to*. The amount of indentation to delete is determined by calling `current-left-margin`. In no case does this function delete non-whitespace. If *from* and *to* are omitted, they default to the whole buffer.

`indent-to-left-margin` [Function]
: This function adjusts the indentation at the beginning of the current line to the value specified by the variable `left-margin`. (That may involve either inserting or deleting whitespace.) This function is value of `indent-line-function` in Paragraph-Indent Text mode.

`left-margin` [User Option]
: This variable specifies the base left margin column. In Fundamental mode, `RET` indents to this column. This variable automatically becomes buffer-local when set in any fashion.

`fill-nobreak-predicate` [User Option]
: This variable gives major modes a way to specify not to break a line at certain places. Its value should be a list of functions. Whenever filling considers breaking the line at a certain place in the buffer, it calls each of these functions with no arguments and with point located at that place. If any of the functions returns non-`nil`, then the line won't be broken there.

33.13 Adaptive Fill Mode

When *Adaptive Fill Mode* is enabled, Emacs determines the fill prefix automatically from the text in each paragraph being filled rather than using a predetermined value. During filling, this fill prefix gets inserted at the start of the second and subsequent lines of the paragraph as described in Section 33.11 [Filling], page 901, and in Section 33.14 [Auto Filling], page 907.

`adaptive-fill-mode` [User Option]
: Adaptive Fill mode is enabled when this variable is non-`nil`. It is `t` by default.

`fill-context-prefix` *from to* [Function]
: This function implements the heart of Adaptive Fill mode; it chooses a fill prefix based on the text between *from* and *to*, typically the start and end of a paragraph. It does this by looking at the first two lines of the paragraph, based on the variables described below.

  Usually, this function returns the fill prefix, a string. However, before doing this, the function makes a final check (not specially mentioned in the following) that a

line starting with this prefix wouldn't look like the start of a paragraph. Should this happen, the function signals the anomaly by returning `nil` instead.

In detail, `fill-context-prefix` does this:

1. It takes a candidate for the fill prefix from the first line—it tries first the function in `adaptive-fill-function` (if any), then the regular expression `adaptive-fill-regexp` (see below). The first non-`nil` result of these, or the empty string if they're both `nil`, becomes the first line's candidate.
2. If the paragraph has as yet only one line, the function tests the validity of the prefix candidate just found. The function then returns the candidate if it's valid, or a string of spaces otherwise. (see the description of `adaptive-fill-first-line-regexp` below).
3. When the paragraph already has two lines, the function next looks for a prefix candidate on the second line, in just the same way it did for the first line. If it doesn't find one, it returns `nil`.
4. The function now compares the two candidate prefixes heuristically: if the non-whitespace characters in the line 2 candidate occur in the same order in the line 1 candidate, the function returns the line 2 candidate. Otherwise, it returns the largest initial substring which is common to both candidates (which might be the empty string).

`adaptive-fill-regexp` [User Option]

Adaptive Fill mode matches this regular expression against the text starting after the left margin whitespace (if any) on a line; the characters it matches are that line's candidate for the fill prefix.

The default value matches whitespace with certain punctuation characters intermingled.

`adaptive-fill-first-line-regexp` [User Option]

Used only in one-line paragraphs, this regular expression acts as an additional check of the validity of the one available candidate fill prefix: the candidate must match this regular expression, or match `comment-start-skip`. If it doesn't, `fill-context-prefix` replaces the candidate with a string of spaces of the same width as it.

The default value of this variable is `"\\`[\t]*\\'"`, which matches only a string of whitespace. The effect of this default is to force the fill prefixes found in one-line paragraphs always to be pure whitespace.

`adaptive-fill-function` [User Option]

You can specify more complex ways of choosing a fill prefix automatically by setting this variable to a function. The function is called with point after the left margin (if any) of a line, and it must preserve point. It should return either that line's fill prefix or `nil`, meaning it has failed to determine a prefix.

33.14 Auto Filling

Auto Fill mode is a minor mode that fills lines automatically as text is inserted. See Section "Auto Fill" in *The GNU Emacs Manual*. This section describes some variables

used by Auto Fill mode. For a description of functions that you can call explicitly to fill and justify existing text, see Section 33.11 [Filling], page 901.

Auto Fill mode also enables the functions that change the margins and justification style to refill portions of the text. See Section 33.12 [Margins], page 905.

`auto-fill-function` [Variable]
: The value of this buffer-local variable should be a function (of no arguments) to be called after self-inserting a character from the table `auto-fill-chars`, see below. It may be `nil`, in which case nothing special is done in that case.

  The value of `auto-fill-function` is `do-auto-fill` when Auto Fill mode is enabled. That is a function whose sole purpose is to implement the usual strategy for breaking a line.

`normal-auto-fill-function` [Variable]
: This variable specifies the function to use for `auto-fill-function`, if and when Auto Fill is turned on. Major modes can set buffer-local values for this variable to alter how Auto Fill works.

`auto-fill-chars` [Variable]
: A char table of characters which invoke `auto-fill-function` when self-inserted—space and newline in most language environments. They have an entry `t` in the table.

`comment-auto-fill-only-comments` [User Option]
: This variable, if non-`nil`, means to fill lines automatically within comments only. More precisely, this means that if a comment syntax was defined for the current buffer, then self-inserting a character outside of a comment will not call `auto-fill-function`.

33.15 Sorting Text

The sorting functions described in this section all rearrange text in a buffer. This is in contrast to the function `sort`, which rearranges the order of the elements of a list (see Section 5.6.3 [Rearrangement], page 90). The values returned by these functions are not meaningful.

`sort-subr` *reverse nextrecfun endrecfun* **&optional** *startkeyfun endkeyfun predicate* [Function]
: This function is the general text-sorting routine that subdivides a buffer into records and then sorts them. Most of the commands in this section use this function.

  To understand how `sort-subr` works, consider the whole accessible portion of the buffer as being divided into disjoint pieces called *sort records*. The records may or may not be contiguous, but they must not overlap. A portion of each sort record (perhaps all of it) is designated as the sort key. Sorting rearranges the records in order by their sort keys.

  Usually, the records are rearranged in order of ascending sort key. If the first argument to the `sort-subr` function, *reverse*, is non-`nil`, the sort records are rearranged in order of descending sort key.

The next four arguments to `sort-subr` are functions that are called to move point across a sort record. They are called many times from within `sort-subr`.

1. *nextrecfun* is called with point at the end of a record. This function moves point to the start of the next record. The first record is assumed to start at the position of point when `sort-subr` is called. Therefore, you should usually move point to the beginning of the buffer before calling `sort-subr`.

   This function can indicate there are no more sort records by leaving point at the end of the buffer.
2. *endrecfun* is called with point within a record. It moves point to the end of the record.
3. *startkeyfun* is called to move point from the start of a record to the start of the sort key. This argument is optional; if it is omitted, the whole record is the sort key. If supplied, the function should either return a non-`nil` value to be used as the sort key, or return `nil` to indicate that the sort key is in the buffer starting at point. In the latter case, *endkeyfun* is called to find the end of the sort key.
4. *endkeyfun* is called to move point from the start of the sort key to the end of the sort key. This argument is optional. If *startkeyfun* returns `nil` and this argument is omitted (or `nil`), then the sort key extends to the end of the record. There is no need for *endkeyfun* if *startkeyfun* returns a non-`nil` value.

The argument *predicate* is the function to use to compare keys. It is called with two arguments, the keys to compare, and should return non-`nil` if the first key should come before the second in the sorting order. What exactly are the key arguments depends on what *startkeyfun* and *endkeyfun* return. If *predicate* is omitted or `nil`, it defaults to `<` if the keys are numbers, to `compare-buffer-substrings` if the keys are cons cells (whose `car` and `cdr` are start and end buffer positions of the key), and to `string<` otherwise (with keys assumed to be strings).

As an example of `sort-subr`, here is the complete function definition for `sort-lines`:

```
;; Note that the first two lines of doc string
;; are effectively one line when viewed by a user.
(defun sort-lines (reverse beg end)
  "Sort lines in region alphabetically;\
 argument means descending order.
Called from a program, there are three arguments:
REVERSE (non-nil means reverse order),\
 BEG and END (region to sort).
The variable `sort-fold-case' determines\
 whether alphabetic case affects
the sort order."
  (interactive "P\nr")
  (save-excursion
    (save-restriction
      (narrow-to-region beg end)
      (goto-char (point-min))
      (let ((inhibit-field-text-motion t))
        (sort-subr reverse 'forward-line 'end-of-line)))))
```

Here `forward-line` moves point to the start of the next record, and `end-of-line` moves point to the end of record. We do not pass the arguments *startkeyfun* and *endkeyfun*, because the entire record is used as the sort key.

The `sort-paragraphs` function is very much the same, except that its `sort-subr` call looks like this:

```
(sort-subr reverse
           (lambda ()
             (while (and (not (eobp))
                         (looking-at paragraph-separate))
               (forward-line 1)))
           'forward-paragraph)
```

Markers pointing into any sort records are left with no useful position after `sort-subr` returns.

`sort-fold-case` [User Option]

If this variable is non-`nil`, `sort-subr` and the other buffer sorting functions ignore case when comparing strings.

`sort-regexp-fields` *reverse record-regexp key-regexp start end* [Command]

This command sorts the region between *start* and *end* alphabetically as specified by *record-regexp* and *key-regexp*. If *reverse* is a negative integer, then sorting is in reverse order.

Alphabetical sorting means that two sort keys are compared by comparing the first characters of each, the second characters of each, and so on. If a mismatch is found, it means that the sort keys are unequal; the sort key whose character is less at the point of first mismatch is the lesser sort key. The individual characters are compared according to their numerical character codes in the Emacs character set.

The value of the *record-regexp* argument specifies how to divide the buffer into sort records. At the end of each record, a search is done for this regular expression, and the text that matches it is taken as the next record. For example, the regular expression '`^.+$`', which matches lines with at least one character besides a newline, would make each such line into a sort record. See Section 35.3 [Regular Expressions], page 999, for a description of the syntax and meaning of regular expressions.

The value of the *key-regexp* argument specifies what part of each record is the sort key. The *key-regexp* could match the whole record, or only a part. In the latter case, the rest of the record has no effect on the sorted order of records, but it is carried along when the record moves to its new position.

The *key-regexp* argument can refer to the text matched by a subexpression of *record-regexp*, or it can be a regular expression on its own.

If *key-regexp* is:

'`\`*digit*' then the text matched by the *digit*th '`\(...\)`' parenthesis grouping in *record-regexp* is the sort key.

'`\&`' then the whole record is the sort key.

a regular expression
: then `sort-regexp-fields` searches for a match for the regular expression within the record. If such a match is found, it is the sort key. If there is no match for *key-regexp* within a record then that record is ignored, which means its position in the buffer is not changed. (The other records may move around it.)

For example, if you plan to sort all the lines in the region by the first word on each line starting with the letter '`f`', you should set *record-regexp* to '`^.*$`' and set *key-regexp* to '`\<f\w*\>`'. The resulting expression looks like this:

```
(sort-regexp-fields nil "^.*$" "\\<f\\w*\\>"
                    (region-beginning)
                    (region-end))
```

If you call `sort-regexp-fields` interactively, it prompts for *record-regexp* and *key-regexp* in the minibuffer.

`sort-lines` *reverse start end* [Command]
: This command alphabetically sorts lines in the region between *start* and *end*. If *reverse* is non-`nil`, the sort is in reverse order.

`sort-paragraphs` *reverse start end* [Command]
: This command alphabetically sorts paragraphs in the region between *start* and *end*. If *reverse* is non-`nil`, the sort is in reverse order.

`sort-pages` *reverse start end* [Command]
: This command alphabetically sorts pages in the region between *start* and *end*. If *reverse* is non-`nil`, the sort is in reverse order.

`sort-fields` *field start end* [Command]
: This command sorts lines in the region between *start* and *end*, comparing them alphabetically by the *field*th field of each line. Fields are separated by whitespace and numbered starting from 1. If *field* is negative, sorting is by the $-$*field*th field from the end of the line. This command is useful for sorting tables.

`sort-numeric-fields` *field start end* [Command]
: This command sorts lines in the region between *start* and *end*, comparing them numerically by the *field*th field of each line. Fields are separated by whitespace and numbered starting from 1. The specified field must contain a number in each line of the region. Numbers starting with 0 are treated as octal, and numbers starting with '`0x`' are treated as hexadecimal.

  If *field* is negative, sorting is by the $-$*field*th field from the end of the line. This command is useful for sorting tables.

`sort-numeric-base` [User Option]
: This variable specifies the default radix for `sort-numeric-fields` to parse numbers.

`sort-columns` *reverse* **&optional** *beg end* [Command]
: This command sorts the lines in the region between *beg* and *end*, comparing them alphabetically by a certain range of columns. The column positions of *beg* and *end* bound the range of columns to sort on.

If *reverse* is non-`nil`, the sort is in reverse order.

One unusual thing about this command is that the entire line containing position *beg*, and the entire line containing position *end*, are included in the region sorted.

Note that `sort-columns` rejects text that contains tabs, because tabs could be split across the specified columns. Use *`M-x untabify`* to convert tabs to spaces before sorting.

When possible, this command actually works by calling the `sort` utility program.

33.16 Counting Columns

The column functions convert between a character position (counting characters from the beginning of the buffer) and a column position (counting screen characters from the beginning of a line).

These functions count each character according to the number of columns it occupies on the screen. This means control characters count as occupying 2 or 4 columns, depending upon the value of `ctl-arrow`, and tabs count as occupying a number of columns that depends on the value of `tab-width` and on the column where the tab begins. See Section 41.23.1 [Usual Display], page 1247.

Column number computations ignore the width of the window and the amount of horizontal scrolling. Consequently, a column value can be arbitrarily high. The first (or leftmost) column is numbered 0. They also ignore overlays and text properties, aside from invisibility. Invisible text is considered as having zero width, unless `buffer-invisibility-spec` specifies that invisible text should be displayed as ellipsis (see Section 41.6 [Invisible Text], page 1146).

`current-column` [Function]

This function returns the horizontal position of point, measured in columns, counting from 0 at the left margin. The column position is the sum of the widths of all the displayed representations of the characters between the start of the current line and point.

`move-to-column` *column* **&optional** *force* [Command]

This function moves point to *column* in the current line. The calculation of *column* takes into account the widths of the displayed representations of the characters between the start of the line and point.

When called interactively, *column* is the value of prefix numeric argument. If *column* is not an integer, an error is signaled.

If it is impossible to move to column *column* because that is in the middle of a multicolumn character such as a tab, point moves to the end of that character. However, if *force* is non-`nil`, and *column* is in the middle of a tab, then `move-to-column` either converts the tab into spaces (when `indent-tabs-mode` is `nil`), or inserts enough spaces before it (otherwise), so that point can move precisely to column *column*. Other multicolumn characters can cause anomalies despite *force*, since there is no way to split them.

The argument *force* also has an effect if the line isn't long enough to reach column *column*; if it is `t`, that means to add whitespace at the end of the line to reach that column.

The return value is the column number actually moved to.

33.17 Indentation

The indentation functions are used to examine, move to, and change whitespace that is at the beginning of a line. Some of the functions can also change whitespace elsewhere on a line. Columns and indentation count from zero at the left margin.

33.17.1 Indentation Primitives

This section describes the primitive functions used to count and insert indentation. The functions in the following sections use these primitives. See Section 41.10 [Size of Displayed Text], page 1161, for related functions.

`current-indentation` [Function]

This function returns the indentation of the current line, which is the horizontal position of the first nonblank character. If the contents are entirely blank, then this is the horizontal position of the end of the line.

This function considers invisible text as having zero width, unless `buffer-invisibility-spec` specifies that invisible text should be displayed as ellipsis. See Section 41.6 [Invisible Text], page 1146.

`indent-to` *column* **&optional** *minimum* [Command]

This function indents from point with tabs and spaces until *column* is reached. If *minimum* is specified and non-`nil`, then at least that many spaces are inserted even if this requires going beyond *column*. Otherwise the function does nothing if point is already beyond *column*. The value is the column at which the inserted indentation ends.

The inserted whitespace characters inherit text properties from the surrounding text (usually, from the preceding text only). See Section 33.19.6 [Sticky Properties], page 935.

`indent-tabs-mode` [User Option]

If this variable is non-`nil`, indentation functions can insert tabs as well as spaces. Otherwise, they insert only spaces. Setting this variable automatically makes it buffer-local in the current buffer.

33.17.2 Indentation Controlled by Major Mode

An important function of each major mode is to customize the `TAB` key to indent properly for the language being edited. This section describes the mechanism of the `TAB` key and how to control it. The functions in this section return unpredictable values.

`indent-for-tab-command` **&optional** *rigid* [Command]

This is the command bound to `TAB` in most editing modes. Its usual action is to indent the current line, but it can alternatively insert a tab character or indent a region.

Here is what it does:

- First, it checks whether Transient Mark mode is enabled and the region is active. If so, it calls `indent-region` to indent all the text in the region (see Section 33.17.3 [Region Indent], page 915).

- Otherwise, if the indentation function in `indent-line-function` is `indent-to-left-margin` (a trivial command that inserts a tab character), or if the variable `tab-always-indent` specifies that a tab character ought to be inserted (see below), then it inserts a tab character.
- Otherwise, it indents the current line; this is done by calling the function in `indent-line-function`. If the line is already indented, and the value of `tab-always-indent` is `complete` (see below), it tries completing the text at point.

If *rigid* is non-`nil` (interactively, with a prefix argument), then after this command indents a line or inserts a tab, it also rigidly indents the entire balanced expression which starts at the beginning of the current line, in order to reflect the new indentation. This argument is ignored if the command indents the region.

`indent-line-function` [Variable]
This variable's value is the function to be used by `indent-for-tab-command`, and various other indentation commands, to indent the current line. It is usually assigned by the major mode; for instance, Lisp mode sets it to `lisp-indent-line`, C mode sets it to `c-indent-line`, and so on. The default value is `indent-relative`. See Section 24.7 [Auto-Indentation], page 582.

`indent-according-to-mode` [Command]
This command calls the function in `indent-line-function` to indent the current line in a way appropriate for the current major mode.

`newline-and-indent` [Command]
This function inserts a newline, then indents the new line (the one following the newline just inserted) according to the major mode. It does indentation by calling `indent-according-to-mode`.

`reindent-then-newline-and-indent` [Command]
This command reindents the current line, inserts a newline at point, and then indents the new line (the one following the newline just inserted). It does indentation on both lines by calling `indent-according-to-mode`.

`tab-always-indent` [User Option]
This variable can be used to customize the behavior of the `TAB` (`indent-for-tab-command`) command. If the value is `t` (the default), the command normally just indents the current line. If the value is `nil`, the command indents the current line only if point is at the left margin or in the line's indentation; otherwise, it inserts a tab character. If the value is `complete`, the command first tries to indent the current line, and if the line was already indented, it calls `completion-at-point` to complete the text at point (see Section 21.6.8 [Completion in Buffers], page 412).

`tab-first-completion` [User Option]
If `tab-always-indent` is `complete`, whether to expand or indent can be further customized via the `tab-first-completion` variable. The following values can be used:

`eol` Only complete if point is at the end of a line.

`word` Complete unless the next character has word syntax.

`word-or-paren`
Complete unless the next character has word syntax or is a parenthesis.

`word-or-paren-or-punct`
Complete unless the next character has word syntax, or is a parenthesis, or is punctuation.

In any case, typing *TAB* a second time always results in completion.

Some major modes need to support embedded regions of text whose syntax belongs to a different major mode. Examples include *literate programming* source files that combine documentation and snippets of source code, Yacc/Bison programs that include snippets of Python or JS code, etc. To correctly indent the embedded chunks, the primary mode needs to delegate the indentation to another mode's indentation engine (e.g., call `js-indent-line` for JS code or `python-indent-line` for Python), while providing it with some context to guide the indentation. Major modes, for their part, should avoid calling `widen` in their indentation code and obey `prog-first-column`.

`prog-indentation-context` [Variable]
This variable, when non-`nil`, holds the indentation context for the sub-mode's indentation engine provided by the superior major mode. The value should be a list of the form `(`*`first-column`* `.` *`rest`*. The members of the list have the following meaning:

first-column
The column to be used for top-level constructs. This replaces the default value of the top-level column used by the sub-mode, usually zero.

rest This value is currently unused.

The following convenience function should be used by major mode's indentation engine in support of invocations as sub-modes of another major mode.

`prog-first-column` [Function]
Call this function instead of using a literal value (usually, zero) of the column number for indenting top-level program constructs. The function's value is the column number to use for top-level constructs. When no superior mode is in effect, this function returns zero.

33.17.3 Indenting an Entire Region

This section describes commands that indent all the lines in the region. They return unpredictable values.

`indent-region` *start end* **&optional** *to-column* [Command]
This command indents each nonblank line starting between *start* (inclusive) and *end* (exclusive). If *to-column* is `nil`, `indent-region` indents each nonblank line by calling the current mode's indentation function, the value of `indent-line-function`.

If *to-column* is non-`nil`, it should be an integer specifying the number of columns of indentation; then this function gives each line exactly that much indentation, by either adding or deleting whitespace.

If there is a fill prefix, `indent-region` indents each line by making it start with the fill prefix.

`indent-region-function` [Variable]
The value of this variable is a function that can be used by `indent-region` as a short cut. It should take two arguments, the start and end of the region. You should design the function so that it will produce the same results as indenting the lines of the region one by one, but presumably faster.

If the value is `nil`, there is no short cut, and `indent-region` actually works line by line.

A short-cut function is useful in modes such as C mode and Lisp mode, where the `indent-line-function` must scan from the beginning of the function definition: applying it to each line would be quadratic in time. The short cut can update the scan information as it moves through the lines indenting them; this takes linear time. In a mode where indenting a line individually is fast, there is no need for a short cut.

`indent-region` with a non-`nil` argument *to-column* has a different meaning and does not use this variable.

`indent-rigidly` *start end count* [Command]
This function indents all lines starting between *start* (inclusive) and *end* (exclusive) sideways by *count* columns. This preserves the shape of the affected region, moving it as a rigid unit.

This is useful not only for indenting regions of unindented text, but also for indenting regions of formatted code. For example, if *count* is 3, this command adds 3 columns of indentation to every line that begins in the specified region.

If called interactively with no prefix argument, this command invokes a transient mode for adjusting indentation rigidly. See Section "Indentation Commands" in *The GNU Emacs Manual*.

`indent-code-rigidly` *start end columns* **&optional** *nochange-regexp* [Command]
This is like `indent-rigidly`, except that it doesn't alter lines that start within strings or comments.

In addition, it doesn't alter a line if *nochange-regexp* matches at the beginning of the line (if *nochange-regexp* is non-`nil`).

33.17.4 Indentation Relative to Previous Lines

This section describes two commands that indent the current line based on the contents of previous lines.

`indent-relative` **&optional** *first-only unindented-ok* [Command]
This command inserts whitespace at point, extending to the same column as the next *indent point* of the previous nonblank line. An indent point is a non-whitespace character following whitespace. The next indent point is the first one at a column greater than the current column of point. For example, if point is underneath and to the left of the first non-blank character of a line of text, it moves to that column by inserting whitespace.

If the previous nonblank line has no next indent point (i.e., none at a great enough column position), `indent-relative` either does nothing (if *unindented-ok* is non-`nil`) or calls `tab-to-tab-stop`. Thus, if point is underneath and to the right of the last column of a short line of text, this command ordinarily moves point to the next tab stop by inserting whitespace.

If *first-only* is non-`nil`, only the first indent point is considered.

The return value of `indent-relative` is unpredictable.

In the following example, point is at the beginning of the second line:

```
                This line is indented twelve spaces.
    ⋆The quick brown fox jumped.
```

Evaluation of the expression `(indent-relative nil)` produces the following:

```
                This line is indented twelve spaces.
                ⋆The quick brown fox jumped.
```

In this next example, point is between the '`m`' and '`p`' of '`jumped`':

```
                This line is indented twelve spaces.
    The quick brown fox jum⋆ped.
```

Evaluation of the expression `(indent-relative nil)` produces the following:

```
                This line is indented twelve spaces.
    The quick brown fox jum  ⋆ped.
```

`indent-relative-first-indent-point` [Command]

This command indents the current line like the previous nonblank line, by calling `indent-relative` with `t` as the *first-only* argument. The return value is unpredictable.

If the previous nonblank line has no indent points beyond the current column, this command does nothing.

33.17.5 Adjustable Tab Stops

This section explains the mechanism for user-specified tab stops and the mechanisms that use and set them. The name "tab stops" is used because the feature is similar to that of the tab stops on a typewriter. The feature works by inserting an appropriate number of spaces and tab characters to reach the next tab stop column; it does not affect the display of tab characters in the buffer (see Section 41.23.1 [Usual Display], page 1247). Note that the `TAB` character as input uses this tab stop feature only in a few major modes, such as Text mode. See Section "Tab Stops" in *The GNU Emacs Manual*.

`tab-to-tab-stop` [Command]

This command inserts spaces or tabs before point, up to the next tab stop column defined by `tab-stop-list`.

`tab-stop-list` [User Option]

This variable defines the tab stop columns used by `tab-to-tab-stop`. It should be either `nil`, or a list of increasing integers, which need not be evenly spaced. The list is implicitly extended to infinity through repetition of the interval between the last and penultimate elements (or `tab-width` if the list has fewer than two elements). A value of `nil` means a tab stop every `tab-width` columns.

Use `M-x edit-tab-stops` to edit the location of tab stops interactively.

33.17.6 Indentation-Based Motion Commands

These commands, primarily for interactive use, act based on the indentation in the text.

`back-to-indentation` [Command]
: This command moves point to the first non-whitespace character in the current line (which is the line in which point is located).

`backward-to-indentation` **&optional** *arg* [Command]
: This command moves point backward *arg* lines and then to the first nonblank character on that line. If *arg* is omitted or `nil`, it defaults to 1.

`forward-to-indentation` **&optional** *arg* [Command]
: This command moves point forward *arg* lines and then to the first nonblank character on that line. If *arg* is omitted or `nil`, it defaults to 1.

33.18 Case Changes

The case change commands described here work on text in the current buffer. See Section 4.9 [Case Conversion], page 73, for case conversion functions that work on strings and characters. See Section 4.10 [Case Tables], page 74, for how to customize which characters are upper or lower case and how to convert them.

`capitalize-region` *start end* [Command]
: This function capitalizes all words in the region defined by *start* and *end*. To capitalize means to convert each word's first character to upper case and convert the rest of each word to lower case. The function returns `nil`.

  If one end of the region is in the middle of a word, the part of the word within the region is treated as an entire word.

  When `capitalize-region` is called interactively, *start* and *end* are point and the mark, with the smallest first.

```
---------- Buffer: foo ----------
This is the contents of the 5th foo.
---------- Buffer: foo ----------

(capitalize-region 1 37)
⇒ nil

---------- Buffer: foo ----------
This Is The Contents Of The 5th Foo.
---------- Buffer: foo ----------
```

`downcase-region` *start end* [Command]
: This function converts all of the letters in the region defined by *start* and *end* to lower case. The function returns `nil`.

  When `downcase-region` is called interactively, *start* and *end* are point and the mark, with the smallest first.

`upcase-region` *start end* [Command]
: This function converts all of the letters in the region defined by *start* and *end* to upper case. The function returns `nil`.

  When `upcase-region` is called interactively, *start* and *end* are point and the mark, with the smallest first.

`capitalize-word` *count* [Command]
: This function capitalizes *count* words after point, moving point over as it does. To capitalize means to convert each word's first character to upper case and convert the rest of each word to lower case. If *count* is negative, the function capitalizes the −*count* previous words but does not move point. The value is `nil`.

  If point is in the middle of a word, the part of the word before point is ignored when moving forward. The rest is treated as an entire word.

  When `capitalize-word` is called interactively, *count* is set to the numeric prefix argument.

`downcase-word` *count* [Command]
: This function converts the *count* words after point to all lower case, moving point over as it does. If *count* is negative, it converts the −*count* previous words but does not move point. The value is `nil`.

  When `downcase-word` is called interactively, *count* is set to the numeric prefix argument.

`upcase-word` *count* [Command]
: This function converts the *count* words after point to all upper case, moving point over as it does. If *count* is negative, it converts the −*count* previous words but does not move point. The value is `nil`.

  When `upcase-word` is called interactively, *count* is set to the numeric prefix argument.

33.19 Text Properties

Each character position in a buffer or a string can have a *text property list*, much like the property list of a symbol (see Section 5.9 [Property Lists], page 99). The properties belong to a particular character at a particular place, such as, the letter '`T`' at the beginning of this sentence or the first '`o`' in '`foo`'—if the same character occurs in two different places, the two occurrences in general have different properties.

Each property has a name and a value. Both of these can be any Lisp object, but the name is normally a symbol. Typically each property name symbol is used for a particular purpose; for instance, the text property `face` specifies the faces for displaying the character (see Section 33.19.4 [Special Properties], page 927). The usual way to access the property list is to specify a name and ask what value corresponds to it.

If a character has a `category` property, we call it the *property category* of the character. It should be a symbol. The properties of the symbol serve as defaults for the properties of the character.

Copying text between strings and buffers preserves the properties along with the characters; this includes such diverse functions as `substring`, `insert`, and `buffer-substring`.

Killing and then yanking text (see Section 33.8 [The Kill Ring], page 891) also preserves the properties, except that some properties are handled specially and might be removed when text is yanked; see Section 33.8.3 [Yanking], page 893.

33.19.1 Examining Text Properties

The simplest way to examine text properties is to ask for the value of a particular property of a particular character. For that, use `get-text-property`. Use `text-properties-at` to get the entire property list of a character. See Section 33.19.3 [Property Search], page 924, for functions to examine the properties of a number of characters at once.

These functions handle both strings and buffers. Keep in mind that positions in a string start from 0, whereas positions in a buffer start from 1. Passing a buffer other than the current buffer may be slow.

`get-text-property` *pos prop* **&optional** *object* [Function]
: This function returns the value of the *prop* property of the character after position *pos* in *object* (a buffer or string). The argument *object* is optional and defaults to the current buffer.

  If *position* is at the end of *object*, the value is `nil`, but note that buffer narrowing does not affect the value. That is, if *object* is a buffer or `nil`, and the buffer is narrowed and *position* is at the end of the narrowed buffer, the result may be non-`nil`.

  If there is no *prop* property strictly speaking, but the character has a property category that is a symbol, then `get-text-property` returns the *prop* property of that symbol.

`get-char-property` *position prop* **&optional** *object* [Function]
: This function is like `get-text-property`, except that it checks overlays first and then text properties. See Section 41.9 [Overlays], page 1154.

  The argument *object* may be a string, a buffer, or a window. If it is a window, then the buffer displayed in that window is used for text properties and overlays, but only the overlays active for that window are considered. If *object* is a buffer, then overlays in that buffer are considered first, in order of decreasing priority, followed by the text properties. If *object* is a string, only text properties are considered, since strings never have overlays.

`get-pos-property` *position prop* **&optional** *object* [Function]
: This function is like `get-char-property`, except that it pays attention to properties' stickiness and overlays' advancement settings instead of the property of the character at (i.e., right after) *position*.

`get-char-property-and-overlay` *position prop* **&optional** *object* [Function]
: This is like `get-char-property`, but gives extra information about the overlay that the property value comes from.

  Its value is a cons cell whose CAR is the property value, the same value `get-char-property` would return with the same arguments. Its CDR is the overlay in which the property was found, or `nil`, if it was found as a text property or not found at all.

  If *position* is at the end of *object*, both the CAR and the CDR of the value are `nil`.

`char-property-alias-alist` [Variable]
: This variable holds an alist which maps property names to a list of alternative property names. If a character does not specify a direct value for a property, the alternative property names are consulted in order; the first non-`nil` value is used. This variable takes precedence over `default-text-properties`, and `category` properties take precedence over this variable.

`text-properties-at` *position* **&optional** *object* [Function]
: This function returns the entire property list of the character at *position* in the string or buffer *object*. If *object* is `nil`, it defaults to the current buffer.

  If *position* is at the end of *object*, the value is `nil`, but note that buffer narrowing does not affect the value. That is, if *object* is a buffer or `nil`, and the buffer is narrowed and *position* is at the end of the narrowed buffer, the result may be non-`nil`.

`default-text-properties` [Variable]
: This variable holds a property list giving default values for text properties. Whenever a character does not specify a value for a property, neither directly, through a category symbol, or through `char-property-alias-alist`, the value stored in this list is used instead. Here is an example:

```
(setq default-text-properties '(foo 69)
      char-property-alias-alist nil)
;; Make sure character 1 has no properties of its own.
(set-text-properties 1 2 nil)
;; What we get, when we ask, is the default value.
(get-text-property 1 'foo)
     ⇒ 69
```

`object-intervals` *OBJECT* [Function]
: This function returns a copy of the intervals (i.e., text properties) in *object* as a list of intervals. *object* must be a string or a buffer. Altering the structure of this list does not change the intervals in the object.

```
(object-intervals (propertize "foo" 'face 'bold))
     ⇒ ((0 3 (face bold)))
```

  Each element in the returned list represents one interval. Each interval has three parts: The first is the start, the second is the end, and the third part is the text property itself.

33.19.2 Changing Text Properties

The primitives for changing properties apply to a specified range of text in a buffer or string. The function `set-text-properties` (see end of section) sets the entire property list of the text in that range; more often, it is useful to add, change, or delete just certain properties specified by name.

Since text properties are considered part of the contents of the buffer (or string), and can affect how a buffer looks on the screen, any change in buffer text properties marks the buffer as modified. Buffer text property changes are undoable also (see Section 33.9 [Undo], page 898). Positions in a string start from 0, whereas positions in a buffer start from 1.

`put-text-property` *start end prop value* **&optional** *object* [Function]
This function sets the *prop* property to *value* for the text between *start* and *end* in the string or buffer *object*. If *object* is `nil`, it defaults to the current buffer.

`add-text-properties` *start end props* **&optional** *object* [Function]
This function adds or overrides text properties for the text between *start* and *end* in the string or buffer *object*. If *object* is `nil`, it defaults to the current buffer.

The argument *props* specifies which properties to add. It should have the form of a property list (see Section 5.9 [Property Lists], page 99): a list whose elements include the property names followed alternately by the corresponding values.

The return value is `t` if the function actually changed some property's value; `nil` otherwise (if *props* is `nil` or its values agree with those in the text).

For example, here is how to set the `comment` and `face` properties of a range of text:

```
(add-text-properties start end
                     '(comment t face highlight))
```

`remove-text-properties` *start end props* **&optional** *object* [Function]
This function deletes specified text properties from the text between *start* and *end* in the string or buffer *object*. If *object* is `nil`, it defaults to the current buffer.

The argument *props* specifies which properties to delete. It should have the form of a property list (see Section 5.9 [Property Lists], page 99): a list whose elements are property names alternating with corresponding values. But only the names matter—the values that accompany them are ignored. For example, here's how to remove the `face` property.

```
(remove-text-properties start end '(face nil))
```

The return value is `t` if the function actually changed some property's value; `nil` otherwise (if *props* is `nil` or if no character in the specified text had any of those properties).

To remove all text properties from certain text, use `set-text-properties` and specify `nil` for the new property list.

`remove-list-of-text-properties` *start end list-of-properties* [Function]
&optional *object*
Like `remove-text-properties` except that *list-of-properties* is a list of property names only, not an alternating list of property names and values.

`set-text-properties` *start end props* **&optional** *object* [Function]
This function completely replaces the text property list for the text between *start* and *end* in the string or buffer *object*. If *object* is `nil`, it defaults to the current buffer.

The argument *props* is the new property list. It should be a list whose elements are property names alternating with corresponding values.

After `set-text-properties` returns, all the characters in the specified range have identical properties.

If *props* is `nil`, the effect is to get rid of all properties from the specified range of text. Here's an example:

```
(set-text-properties start end nil)
```

Do not rely on the return value of this function.

`add-face-text-property` *start end face* **&optional** *appendp object* [Function]
This function acts on the text between *start* and *end*, adding the face *face* to the `face` text property. *face* should be a valid value for the `face` property (see Section 33.19.4 [Special Properties], page 927), such as a face name or an anonymous face (see Section 41.12 [Faces], page 1168).

If any text in the region already has a non-`nil` `face` property, those face(s) are retained. This function sets the `face` property to a list of faces, with *face* as the first element (by default) and the pre-existing faces as the remaining elements. If the optional argument *appendp* is non-`nil`, *face* is appended to the end of the list instead. Note that in a face list, the first occurring value for each attribute takes precedence.

For example, the following code would assign an italicized green face to the text between *start* and *end*:

```
(add-face-text-property start end 'italic)
(add-face-text-property start end '(:foreground "red"))
(add-face-text-property start end '(:foreground "green"))
```

The optional argument *object*, if non-`nil`, specifies a buffer or string to act on, rather than the current buffer. If *object* is a string, then *start* and *end* are zero-based indices into the string.

The easiest way to make a string with text properties is with `propertize`:

`propertize` *string* **&rest** *properties* [Function]
This function returns a copy of *string* with the text properties *properties* added. These properties apply to all the characters in the string that is returned. Here is an example that constructs a string with a `face` property and a `mouse-face` property:

```
(propertize "foo" 'face 'italic
            'mouse-face 'bold-italic)
     ⇒ #("foo" 0 3 (mouse-face bold-italic face italic))
```

To put different properties on various parts of a string, you can construct each part with `propertize` and then combine them with `concat`:

```
(concat
 (propertize "foo" 'face 'italic
             'mouse-face 'bold-italic)
 " and "
 (propertize "bar" 'face 'italic
             'mouse-face 'bold-italic))
     ⇒ #("foo and bar"
                 0 3 (face italic mouse-face bold-italic)
                 3 8 nil
                 8 11 (face italic mouse-face bold-italic))
```

See Section 33.2 [Buffer Contents], page 881, for the function `buffer-substring-no-properties`, which copies text from the buffer but does not copy its properties.

If you wish to add text properties to a buffer or remove them without marking the buffer as modified, you can wrap the calls above in the `with-silent-modifications` macro. See Section 28.5 [Buffer Modification], page 679.

33.19.3 Text Property Search Functions

In typical use of text properties, most of the time several or many consecutive characters have the same value for a property. Rather than writing your programs to examine characters one by one, it is much faster to process chunks of text that have the same property value.

Here are functions you can use to do this. They use `eq` for comparing property values. In all cases, *object* defaults to the current buffer.

For good performance, it's very important to use the *limit* argument to these functions, especially the ones that search for a single property—otherwise, they may spend a long time scanning to the end of the buffer, if the property you are interested in does not change.

These functions do not move point; instead, they return a position (or `nil`). Remember that a position is always between two characters; the position returned by these functions is between two characters with different properties.

`next-property-change` *pos* **&optional** *object limit* [Function]

The function scans the text forward from position *pos* in the string or buffer *object* until it finds a change in some text property, then returns the position of the change. In other words, it returns the position of the first character beyond *pos* whose properties are not identical to those of the character just after *pos*.

If *limit* is non-`nil`, then the scan ends at position *limit*. If there is no property change before that point, this function returns *limit*.

The value is `nil` if the properties remain unchanged all the way to the end of *object* and *limit* is `nil`. If the value is non-`nil`, it is a position greater than or equal to *pos*. The value equals *pos* only when *limit* equals *pos*.

Here is an example of how to scan the buffer by chunks of text within which all properties are constant:

```
(while (not (eobp))
  (let ((plist (text-properties-at (point)))
        (next-change
         (or (next-property-change (point) (current-buffer))
             (point-max))))
    Process text from point to next-change...
    (goto-char next-change)))
```

`previous-property-change` *pos* **&optional** *object limit* [Function]

This is like `next-property-change`, but scans back from *pos* instead of forward. If the value is non-`nil`, it is a position less than or equal to *pos*; it equals *pos* only if *limit* equals *pos*.

`next-single-property-change` *pos prop* **&optional** *object limit* [Function]

The function scans text for a change in the *prop* property, then returns the position of the change. The scan goes forward from position *pos* in the string or buffer *object*. In other words, this function returns the position of the first character beyond *pos* whose *prop* property differs from that of the character just after *pos*.

If *limit* is non-`nil`, then the scan ends at position *limit*. If there is no property change before that point, `next-single-property-change` returns *limit*.

The value is `nil` if the property remains unchanged all the way to the end of *object* and *limit* is `nil`. If the value is non-`nil`, it is a position greater than or equal to *pos*; it equals *pos* only if *limit* equals *pos*.

`previous-single-property-change` *pos prop* **&optional** *object limit* [Function]
This is like `next-single-property-change`, but scans back from *pos* instead of forward. If the value is non-`nil`, it is a position less than or equal to *pos*; it equals *pos* only if *limit* equals *pos*.

`next-char-property-change` *pos* **&optional** *limit* [Function]
This is like `next-property-change` except that it considers overlay properties as well as text properties, and if no change is found before the end of the buffer, it returns the maximum buffer position rather than `nil` (in this sense, it resembles the corresponding overlay function `next-overlay-change`, rather than `next-property-change`). There is no *object* operand because this function operates only on the current buffer. It returns the next address at which either kind of property changes.

`previous-char-property-change` *pos* **&optional** *limit* [Function]
This is like `next-char-property-change`, but scans back from *pos* instead of forward, and returns the minimum buffer position if no change is found.

`next-single-char-property-change` *pos prop* **&optional** *object limit* [Function]
This is like `next-single-property-change` except that it considers overlay properties as well as text properties, and if no change is found before the end of the *object*, it returns the maximum valid position in *object* rather than `nil`. Unlike `next-char-property-change`, this function *does* have an *object* operand; if *object* is not a buffer, only text-properties are considered.

`previous-single-char-property-change` *pos prop* **&optional** *object limit* [Function]
This is like `next-single-char-property-change`, but scans back from *pos* instead of forward, and returns the minimum valid position in *object* if no change is found.

`text-property-any` *start end prop value* **&optional** *object* [Function]
This function returns non-`nil` if at least one character between *start* and *end* has a property *prop* whose value is *value*. More precisely, it returns the position of the first such character. Otherwise, it returns `nil`.

The optional fifth argument, *object*, specifies the string or buffer to scan. Positions are relative to *object*. The default for *object* is the current buffer.

`text-property-not-all` *start end prop value* **&optional** *object* [Function]
This function returns non-`nil` if at least one character between *start* and *end* does not have a property *prop* with value *value*. More precisely, it returns the position of the first such character. Otherwise, it returns `nil`.

The optional fifth argument, *object*, specifies the string or buffer to scan. Positions are relative to *object*. The default for *object* is the current buffer.

`text-property-search-forward` *prop* **&optional** *value predicate not-current* [Function]

Search for the next region of text whose property *prop* is a match for *value* (which defaults to `nil`), according to *predicate*.

This function is modeled after `search-forward` (see Section 35.1 [String Search], page 996) and friends, in that it moves point, but it also returns a structure that describes the match instead of returning it in `match-beginning` and friends.

If the text property whose value is a match can't be found, the function returns `nil`. If it's found, point is placed at the end of the region that has this matching text property, and the function returns a `prop-match` structure with information about the match.

predicate can either be `t` (which is a synonym for `equal`), `nil` (which means "not equal"), or a predicate that will be called with two arguments: *value* and the value of the text property *prop* at the buffer position that is a candidate for a match. The function should return non-`nil` if there's a match, `nil` otherwise.

If *not-current* is non-`nil`, then if point is already in a region where we have a property match, skip past that region and find the next region instead.

The `prop-match` structure has the following accessor functions: `prop-match-beginning` (the start of the match), `prop-match-end` (the end of the match), and `prop-match-value` (the value of *property* at the start of the match).

In the examples below, we use a buffer whose contents is:

> This is a **bold** and here's *bolditalic* and this is the end.

That is, the "bold" words are the `bold` face, and the "italic" word is in the `italic` face.

With point at the start:

```
(while (setq match (text-property-search-forward 'face 'bold t))
  (push (buffer-substring (prop-match-beginning match)
                          (prop-match-end match))
        words))
```

This will pick out all the words that use the `bold` face.

```
(while (setq match (text-property-search-forward 'face nil t))
  (push (buffer-substring (prop-match-beginning match)
                          (prop-match-end match))
        words))
```

This will pick out all the bits that have no face properties, which will result in the list '`("This is a " "and here's " "and this is the end")`' (only in reverse order, since we used `push`, see Section 5.5 [List Variables], page 85).

```
(while (setq match (text-property-search-forward 'face nil nil))
  (push (buffer-substring (prop-match-beginning match)
                          (prop-match-end match))
        words))
```

This will pick out all the regions where `face` is set to something, but this is split up into where the properties change, so the result here will be '`("bold" "bold" "italic")`'.

For a more realistic example where you might use this, consider that you have a buffer where certain sections represent URLs, and these are tagged with `shr-url`.

```
(while (setq match (text-property-search-forward 'shr-url nil nil))
  (push (prop-match-value match) urls))
```

This will give you a list of all those URLs.

`text-property-search-backward` *prop* **&optional** *value predicate not-current* [Function]

This is just like `text-property-search-forward`, but searches backward instead, and if a match is found, point is placed at the beginning of the matched region instead of the end.

33.19.4 Properties with Special Meanings

Here is a table of text property names that have special built-in meanings. The following sections list a few additional special property names that control filling and property inheritance. All other names have no standard meaning, and you can use them as you like.

Note: the properties `composition`, `display`, `invisible` and `intangible` can also cause point to move to an acceptable place, after each Emacs command. See Section 22.6 [Adjusting Point], page 440.

`category` If a character has a `category` property, we call it the *property category* of the character. It should be a symbol. The properties of this symbol serve as defaults for the properties of the character.

`face` The `face` property controls the appearance of the character (see Section 41.12 [Faces], page 1168). The value of the property can be the following:

- A face name (a symbol or string).
- An anonymous face: a property list of the form `(`*`keyword value ...`*`)`, where each *keyword* is a face attribute name and *value* is a value for that attribute.
- A list of faces. Each list element should be either a face name or an anonymous face. This specifies a face which is an aggregate of the attributes of each of the listed faces. Faces occurring earlier in the list have higher priority.
- A cons cell of the form `(foreground-color .` *`color-name`*`)` or `(background-color .` *`color-name`*`)`. This specifies the foreground or background color, similar to `(:foreground` *`color-name`*`)` or `(:background` *`color-name`*`)`. This form is supported for backward compatibility only, and should be avoided.
- A cons cell of the form `(:filtered` *`filter face-spec`*`)`, that specifies the face given by *face-spec*, but only if *filter* matches when the face is used for display. The *face-spec* can use any of the forms mentioned above. The *filter* should be of the form `(:window` *`param value`*`)`, which matches for windows whose parameter *param* is `eq` to *value*. If the variable `face-filters-always-match` is non-`nil`, all face filters are deemed to have matched.

Font Lock mode (see Section 24.6 [Font Lock Mode], page 564) works in most buffers by dynamically updating the `face` property of characters based on the context.

The `add-face-text-property` function provides a convenient way to set this text property. See Section 33.19.2 [Changing Properties], page 921.

`font-lock-face`
: This property specifies a value for the `face` property that Font Lock mode should apply to the underlying text. It is one of the fontification methods used by Font Lock mode, and is useful for special modes that implement their own highlighting. See Section 24.6.6 [Precalculated Fontification], page 574. When Font Lock mode is disabled, `font-lock-face` has no effect.

`mouse-face`
: This property is used instead of `face` when the mouse pointer hovers over the text which has this property. When this happens, the entire stretch of text that has the same `mouse-face` property value, not just the character under the mouse, is highlighted.

  Emacs ignores all face attributes from the `mouse-face` property that alter the text size (e.g., `:height`, `:weight`, and `:slant`). Those attributes are always the same as for the unhighlighted text.

`cursor-face`
: This property is similar to `mouse-face`, but it is used when point (not the mouse) is inside text that has this property. The highlighting happens only if the mode `cursor-face-highlight-mode` is enabled. When the variable `cursor-face-highlight-nonselected-window` is non-`nil`, the text with this face is highlighted even if the window is not selected, similarly to what `highlight-nonselected-windows` does for the region (see Section "The Mark and the Region" in *The GNU Emacs Manual*).

`fontified`
: This property says whether the text is ready for display. If `nil`, Emacs's redisplay routine calls the functions in `fontification-functions` (see Section 41.12.7 [Auto Faces], page 1182) to prepare this part of the buffer before it is displayed. It is used internally by the just-in-time font locking code.

`display`
: This property activates various features that change the way text is displayed. For example, it can make text appear taller or shorter, higher or lower, wider or narrow, or replaced with an image. See Section 41.16 [Display Property], page 1203.

`help-echo`
: If text has a string as its `help-echo` property, then when you move the mouse onto that text, Emacs displays that string in the echo area, or in the tooltip window (see Section 41.26 [Tooltips], page 1254), after passing it through `substitute-command-keys`.

  If the value of the `help-echo` property is a function, that function is called with three arguments, *window*, *object* and *pos* and should return a help string or `nil` for none. The first argument, *window* is the window in which the help

was found. The second, *object*, is the buffer, overlay or string which had the `help-echo` property. The *pos* argument is as follows:

- If *object* is a buffer, *pos* is the position in the buffer.
- If *object* is an overlay, that overlay has a `help-echo` property, and *pos* is the position in the overlay's buffer.
- If *object* is a string (an overlay string or a string displayed with the `display` property), *pos* is the position in that string.

If the value of the `help-echo` property is neither a function nor a string, it is evaluated to obtain a help string.

You can alter the way help text is displayed by setting the variable `show-help-function` (see [Help display], page 934).

This feature is used in the mode line and for other active text.

`help-echo-inhibit-substitution`
: If the first character of a `help-echo` string has a non-`nil` `help-echo-inhibit-substitution` property, then it is displayed as-is by `show-help-function`, without being passed through `substitute-command-keys`.

`keymap` The `keymap` property specifies an additional keymap for commands. When this keymap applies, it is used for key lookup before the minor mode keymaps and before the buffer's local map. See Section 23.7 [Active Keymaps], page 490. If the property value is a symbol, the symbol's function definition is used as the keymap.

The property's value for the character before point applies if it is non-`nil` and rear-sticky, and the property's value for the character after point applies if it is non-`nil` and front-sticky. (For mouse clicks, the position of the click is used instead of the position of point.)

`local-map`
: This property works like `keymap` except that it specifies a keymap to use *instead of* the buffer's local map. For most purposes (perhaps all purposes), it is better to use the `keymap` property.

`syntax-table`
: The `syntax-table` property overrides what the syntax table says about this particular character. See Section 36.4 [Syntax Properties], page 1032.

`read-only`
: If a character has the property `read-only`, then modifying that character is not allowed. Any command that would do so gets an error, `text-read-only`. If the property value is a string, that string is used as the error message.

Insertion next to a read-only character is an error if inserting ordinary text there would inherit the `read-only` property due to stickiness. Thus, you can control permission to insert next to read-only text by controlling the stickiness. See Section 33.19.6 [Sticky Properties], page 935.

Since changing properties counts as modifying the buffer, it is not possible to remove a `read-only` property unless you know the special trick: bind

`inhibit-read-only` to a non-`nil` value and then remove the property. See Section 28.7 [Read Only Buffers], page 682.

`inhibit-read-only`
: Characters that have the property `inhibit-read-only` can be edited even in read-only buffers. See Section 28.7 [Read Only Buffers], page 682.

`invisible`
: A non-`nil` `invisible` property can make a character invisible on the screen. See Section 41.6 [Invisible Text], page 1146, for details.

`inhibit-isearch`
: A non-`nil` `inhibit-isearch` property will make isearch skip the text.

`intangible`
: If a group of consecutive characters have equal and non-`nil` `intangible` properties, then you cannot place point between them. If you try to move point forward into the group, point actually moves to the end of the group. If you try to move point backward into the group, point actually moves to the start of the group.

  If consecutive characters have unequal non-`nil` `intangible` properties, they belong to separate groups; each group is separately treated as described above.

  When the variable `inhibit-point-motion-hooks` is non-`nil` (as it is by default), the `intangible` property is ignored.

  Beware: this property operates at a very low level, and affects a lot of code in unexpected ways. So use it with extreme caution. A common misuse is to put an intangible property on invisible text, which is actually unnecessary since the command loop will move point outside of the invisible text at the end of each command anyway. See Section 22.6 [Adjusting Point], page 440. For these reasons, this property is obsolete; use the `cursor-intangible` property instead.

`cursor-intangible`
: When the minor mode `cursor-intangible-mode` is turned on, point is moved away from any position that has a non-`nil` `cursor-intangible` property, just before redisplay happens. Note that "stickiness" of the property (see Section 33.19.6 [Sticky Properties], page 935) is taken into account when computing allowed cursor positions, so (for instance) to insert a stretch of five '`x`' characters into which the cursor can't enter, you should do something like:

  ```
  (insert
   (propertize "xxxx" 'cursor-intangible t)
   (propertize "x" 'cursor-intangible t 'rear-nonsticky t))
  ```

  When the variable `cursor-sensor-inhibit` is non-`nil`, the `cursor-intangible` property and the `cursor-sensor-functions` property (described below) are ignored.

`field`
: Consecutive characters with the same `field` property constitute a *field*. Some motion functions including `forward-word` and `beginning-of-line` stop moving at a field boundary. See Section 33.19.9 [Fields], page 939.

`cursor` Normally, the cursor is displayed at the beginning or the end of any overlay and text property strings that "hide" (i.e., are displayed instead of) the current buffer position. You can instead tell Emacs to place the cursor on any desired character of these strings by giving that character a non-`nil` `cursor` text property. In addition, if the value of the `cursor` property is an integer, it specifies the number of buffer's character positions, starting with the position where the overlay or the `display` property begins, for which the cursor should be displayed on that character. Specifically, if the value of the `cursor` property of a character is the number *n*, the cursor will be displayed on this character for any buffer position in the range `[`*ovpos*`..`*ovpos*`+`*n*`)`, where *ovpos* is the overlay's starting position given by `overlay-start` (see Section 41.9.1 [Managing Overlays], page 1154), or the position where the `display` text property begins in the buffer.

In other words, the string character with the `cursor` property of any non-`nil` value is the character where to display the cursor when the overlay or display string make point not visible on display. The value of the property says for which buffer positions to display the cursor there. If the value is an integer *n*, the cursor is displayed there when point is anywhere between the beginning of the overlay or `display` property and *n* positions after that. If the value is anything else and non-`nil`, the cursor is displayed there only when point is at the buffer position that is the beginning of the `display` property, or at `overlay-start` if that position is not visible on display. Note that an integer value of the `cursor` property could mean that the cursor is displayed on that character even when point is visible on display.

One subtlety of this property is that it doesn't work to put this property on a newline character that is part of a display or overlay string. That's because the newline doesn't have a graphic representation on the screen for Emacs to find when it looks for a character on display with that `cursor` property.

When the buffer has many overlay strings (e.g., see Section 41.9.2 [Overlay Properties], page 1157) that conceal some of the buffer text or `display` properties that are strings, it is a good idea to use the `cursor` property on these strings to cue the Emacs display about the places where to put the cursor while traversing these strings. This directly communicates to the display engine where the Lisp program wants to put the cursor, or where the user would expect the cursor, when point is located on some buffer position that is "covered" by the display or overlay string.

`pointer` This specifies a specific pointer shape when the mouse pointer is over this text or image. See Section 30.19 [Pointer Shape], page 841, for possible pointer shapes.

`line-spacing`
A newline can have a `line-spacing` text or overlay property that controls the height of the display line ending with that newline. The property value overrides the default frame line spacing and the buffer local `line-spacing` variable. See Section 41.11 [Line Height], page 1166.

`line-height`
: A newline can have a `line-height` text or overlay property that controls the total height of the display line ending in that newline. See Section 41.11 [Line Height], page 1166.

`wrap-prefix`
: If text has a `wrap-prefix` property, the prefix it defines will be added at display time to the beginning of every continuation line due to text wrapping (so if lines are truncated, the wrap-prefix is never used). It may be a string or an image (see Section 41.16.4 [Other Display Specs], page 1207), or a stretch of whitespace such as specified by the `:width` or `:align-to` display properties (see Section 41.16.2 [Specified Space], page 1205).

  A wrap-prefix may also be specified for an entire buffer using the `wrap-prefix` buffer-local variable (however, a `wrap-prefix` text-property takes precedence over the value of the `wrap-prefix` variable). See Section 41.3 [Truncation], page 1134.

`line-prefix`
: If text has a `line-prefix` property, the prefix it defines will be added at display time to the beginning of every non-continuation line. It may be a string or an image (see Section 41.16.4 [Other Display Specs], page 1207), or a stretch of whitespace such as specified by the `:width` or `:align-to` display properties (see Section 41.16.2 [Specified Space], page 1205).

  A line-prefix may also be specified for an entire buffer using the `line-prefix` buffer-local variable (however, a `line-prefix` text-property takes precedence over the value of the `line-prefix` variable). See Section 41.3 [Truncation], page 1134.

`modification-hooks`
: If a character has the property `modification-hooks`, then its value should be a list of functions; modifying that character calls all of those functions before the actual modification. Each function receives two arguments: the beginning and end of the part of the buffer being modified. Note that if a particular modification hook function appears on several characters being modified by a single primitive, you can't predict how many times the function will be called. Furthermore, insertion will not modify any existing character, so this hook will only be run when removing some characters, replacing them with others, or changing their text-properties.

  Unlike with other similar hooks, when Emacs calls these functions, `inhibit-modification-hooks` does *not* get bound to non-`nil`. If the functions modify the buffer, you should consider binding this variable to non-`nil` to prevent any buffer changes running the change hooks. Otherwise, you must be prepared for recursive calls. See Section 33.34 [Change Hooks], page 964.

  Overlays also support the `modification-hooks` property, but the details are somewhat different (see Section 41.9.2 [Overlay Properties], page 1157).

`insert-in-front-hooks`
`insert-behind-hooks`
: The operation of inserting text in a buffer also calls the functions listed in the `insert-in-front-hooks` property of the following character and in the `insert-behind-hooks` property of the preceding character. These functions receive two arguments, the beginning and end of the inserted text. The functions are called *after* the actual insertion takes place.

  When these functions are called, `inhibit-modification-hooks` is bound to non-`nil`. If the functions modify the buffer, you might want to bind `inhibit-modification-hooks` to `nil`, so as to cause the change hooks to run for these modifications. However, doing this may call your own change hook recursively, so be sure to prepare for that.

  See also Section 33.34 [Change Hooks], page 964, for other hooks that are called when you change text in a buffer.

`point-entered`
`point-left`
: The special properties `point-entered` and `point-left` record hook functions that report motion of point. Each time point moves, Emacs compares these two property values:

  - the `point-left` property of the character after the old location, and
  - the `point-entered` property of the character after the new location.

  If these two values differ, each of them is called (if not `nil`) with two arguments: the old value of point, and the new one.

  The same comparison is made for the characters before the old and new locations. The result may be to execute two `point-left` functions (which may be the same function) and/or two `point-entered` functions (which may be the same function). In any case, all the `point-left` functions are called first, followed by all the `point-entered` functions.

  It is possible to use `char-after` to examine characters at various buffer positions without moving point to those positions. Only an actual change in the value of point runs these hook functions.

  The variable `inhibit-point-motion-hooks` by default inhibits running the `point-left` and `point-entered` hooks, see [Inhibit point motion hooks], page 934.

  These properties are obsolete; please use `cursor-sensor-functions` instead.

`cursor-sensor-functions`
: This special property records a list of functions that react to cursor motion. Each function in the list is called, just before redisplay, with 3 arguments: the affected window, the previous known position of the cursor, and one of the symbols `entered` or `left`, depending on whether the cursor is entering the text that has this property or leaving it. The functions are called only when the minor mode `cursor-sensor-mode` is turned on.

  When the variable `cursor-sensor-inhibit` is non-`nil`, the `cursor-sensor-functions` property is ignored.

`composition`
: This text property is used to display a sequence of characters as a single glyph composed from components. But the value of the property itself is completely internal to Emacs and should not be manipulated directly by, for instance, `put-text-property`.

`minibuffer-message`
: This text property tells where to display temporary messages in an active minibuffer. Specifically, the first character of the minibuffer text which has this property will have the temporary message displayed before it. The default is to display temporary messages at the end of the minibuffer text. This text property is used by the function that is the default value of `set-message-function` (see Section 41.4.1 [Displaying Messages], page 1135).

`inhibit-point-motion-hooks` [Variable]
: When this obsolete variable is non-`nil`, `point-left` and `point-entered` hooks are not run, and the `intangible` property has no effect. Do not set this variable globally; bind it with `let`. Since the affected properties are obsolete, this variable's default value is `t`, to effectively disable them.

`show-help-function` [Variable]
: If this variable is non-`nil`, it specifies a function called to display help strings. These may be `help-echo` properties, menu help strings (see Section 23.18.1.1 [Simple Menu Items], page 511, see Section 23.18.1.2 [Extended Menu Items], page 512), or tool bar help strings (see Section 23.18.6 [Tool Bar], page 518). The specified function is called with one argument, the help string to display, which is passed through `substitute-command-keys` before being given to the function, unless the help string has a non-`nil` `help-echo-inhibit-substitution` property on its first character; see Section 25.3 [Keys in Documentation], page 600. See the code of Tooltip mode (see Section "Tooltips" in *The GNU Emacs Manual*) for an example of a mode that uses `show-help-function`.

`face-filters-always-match` [Variable]
: If this variable is non-`nil`, face filters that specify attributes applied only when certain conditions are met will be deemed to match always.

33.19.5 Formatted Text Properties

These text properties affect the behavior of the fill commands. They are used for representing formatted text. See Section 33.11 [Filling], page 901, and Section 33.12 [Margins], page 905.

`hard`
: If a newline character has this property, it is a "hard" newline. The fill commands do not alter hard newlines and do not move words across them. However, this property takes effect only if the `use-hard-newlines` minor mode is enabled. See Section "Hard and Soft Newlines" in *The GNU Emacs Manual*.

`right-margin`
: This property specifies an extra right margin for filling this part of the text.

`left-margin`
: This property specifies an extra left margin for filling this part of the text.

`justification`
: This property specifies the style of justification for filling this part of the text.

33.19.6 Stickiness of Text Properties

Self-inserting characters, the ones that get inserted into a buffer when the user types them (see Section 33.5 [Commands for Insertion], page 886), normally take on the same properties as the preceding character. This is called *inheritance* of properties.

By contrast, a Lisp program can do insertion with inheritance or without, depending on the choice of insertion primitive. The ordinary text insertion functions, such as `insert`, do not inherit any properties. They insert text with precisely the properties of the string being inserted, and no others. This is correct for programs that copy text from one context to another—for example, into or out of the kill ring. To insert with inheritance, use the special primitives described in this section. Self-inserting characters inherit properties because they work using these primitives.

When you do insertion with inheritance, *which* properties are inherited, and from where, depends on which properties are *sticky*. Insertion after a character inherits those of its properties that are *rear-sticky*. Insertion before a character inherits those of its properties that are *front-sticky*. When both sides offer different sticky values for the same property, the previous character's value takes precedence.

By default, a text property is rear-sticky but not front-sticky; thus, the default is to inherit all the properties of the preceding character, and nothing from the following character.

You can control the stickiness of various text properties with two specific text properties, `front-sticky` and `rear-nonsticky`, and with the variable `text-property-default-nonsticky`. You can use the variable to specify a different default for a given property. You can use those two text properties to make any specific properties sticky or nonsticky in any particular part of the text.

If a character's `front-sticky` property is `t`, then all its properties are front-sticky. If the `front-sticky` property is a list, then the sticky properties of the character are those whose names are in the list. For example, if a character has a `front-sticky` property whose value is `(face read-only)`, then insertion before the character can inherit its `face` property and its `read-only` property, but no others.

The `rear-nonsticky` property works the opposite way. Most properties are rear-sticky by default, so the `rear-nonsticky` property says which properties are *not* rear-sticky. If a character's `rear-nonsticky` property is `t`, then none of its properties are rear-sticky. If the `rear-nonsticky` property is a list, properties are rear-sticky *unless* their names are in the list.

`text-property-default-nonsticky` [Variable]
: This variable holds an alist which defines the default rear-stickiness of various text properties. Each element has the form `(`*`property`*` . `*`nonstickiness`*`)`, and it defines the stickiness of a particular text property, *property*.

  If *nonstickiness* is non-`nil`, this means that the property *property* is rear-nonsticky by default. Since all properties are front-nonsticky by default, this makes *property* nonsticky in both directions by default.

The text properties `front-sticky` and `rear-nonsticky`, when used, take precedence over the default *nonstickiness* specified in `text-property-default-nonsticky`.

Here are the functions that insert text with inheritance of properties:

`insert-and-inherit` **&rest** *strings* [Function]
Insert the strings *strings*, just like the function `insert`, but inherit any sticky properties from the adjoining text.

`insert-before-markers-and-inherit` **&rest** *strings* [Function]
Insert the strings *strings*, just like the function `insert-before-markers`, but inherit any sticky properties from the adjoining text.

See Section 33.4 [Insertion], page 884, for the ordinary insertion functions which do not inherit.

33.19.7 Lazy Computation of Text Properties

Instead of computing text properties for all the text in the buffer, you can arrange to compute the text properties for parts of the text when and if something depends on them.

The primitive that extracts text from the buffer along with its properties is `buffer-substring`. Before examining the properties, this function runs the abnormal hook `buffer-access-fontify-functions`.

`buffer-access-fontify-functions` [Variable]
This variable holds a list of functions for computing text properties. Before `buffer-substring` copies the text and text properties for a portion of the buffer, it calls all the functions in this list. Each of the functions receives two arguments that specify the range of the buffer being accessed. (The buffer itself is always the current buffer.)

The function `buffer-substring-no-properties` does not call these functions, since it ignores text properties anyway.

In order to prevent the hook functions from being called more than once for the same part of the buffer, you can use the variable `buffer-access-fontified-property`.

`buffer-access-fontified-property` [Variable]
If this variable's value is non-`nil`, it is a symbol which is used as a text property name. A non-`nil` value for that text property means the other text properties for this character have already been computed.

If all the characters in the range specified for `buffer-substring` have a non-`nil` value for this property, `buffer-substring` does not call the `buffer-access-fontify-functions` functions. It assumes these characters already have the right text properties, and just copies the properties they already have.

The normal way to use this feature is that the `buffer-access-fontify-functions` functions add this property, as well as others, to the characters they operate on. That way, they avoid being called over and over for the same text.

33.19.8 Defining Clickable Text

Clickable text is text that can be clicked, with either the mouse or via a keyboard command, to produce some result. Many major modes use clickable text to implement textual hyperlinks, or *links* for short.

The easiest way to insert and manipulate links is to use the `button` package. See Section 41.20 [Buttons], page 1236. In this section, we will explain how to manually set up clickable text in a buffer, using text properties. For simplicity, we will refer to the clickable text as a *link*.

Implementing a link involves three separate steps: (1) indicating clickability when the mouse moves over the link; (2) making `RET` or *`mouse-2`* on that link do something; and (3) setting up a `follow-link` condition so that the link obeys `mouse-1-click-follows-link`.

To indicate clickability, add the `mouse-face` text property to the text of the link; then Emacs will highlight the link when the mouse moves over it. In addition, you should define a tooltip or echo area message, using the `help-echo` text property. See Section 33.19.4 [Special Properties], page 927. For instance, here is how Dired indicates that file names are clickable:

```
(if (dired-move-to-filename)
    (add-text-properties
      (point)
      (save-excursion
        (dired-move-to-end-of-filename)
        (point))
      '(mouse-face highlight
        help-echo "mouse-2: visit this file in other window")))
```

To make the link clickable, bind `RET` and *`mouse-2`* to commands that perform the desired action. Each command should check to see whether it was called on a link, and act accordingly. For instance, Dired's major mode keymap binds *`mouse-2`* to the following command:

```
(defun dired-mouse-find-file-other-window (event)
  "In Dired, visit the file or directory name you click on."
  (interactive "e")
  (let ((window (posn-window (event-end event)))
        (pos (posn-point (event-end event)))
        file)
    (if (not (windowp window))
        (error "No file chosen"))
    (with-current-buffer (window-buffer window)
      (goto-char pos)
      (setq file (dired-get-file-for-visit)))
    (if (file-directory-p file)
        (or (and (cdr dired-subdir-alist)
                 (dired-goto-subdir file))
            (progn
              (select-window window)
              (dired-other-window file)))
      (select-window window)
      (find-file-other-window (file-name-sans-versions file t)))))
```

This command uses the functions `posn-window` and `posn-point` to determine where the click occurred, and `dired-get-file-for-visit` to determine which file to visit.

Instead of binding the mouse command in a major mode keymap, you can bind it within the link text, using the `keymap` text property (see Section 33.19.4 [Special Properties], page 927). For instance:

```
(let ((map (make-sparse-keymap)))
  (define-key map [mouse-2] 'operate-this-button)
  (put-text-property link-start link-end 'keymap map))
```

With this method, you can easily define different commands for different links. Furthermore, the global definition of `RET` and *`mouse-2`* remain available for the rest of the text in the buffer.

The basic Emacs command for clicking on links is *`mouse-2`*. However, for compatibility with other graphical applications, Emacs also recognizes *`mouse-1`* clicks on links, provided the user clicks on the link quickly without moving the mouse. This behavior is controlled by the user option `mouse-1-click-follows-link`. See Section "Mouse References" in *The GNU Emacs Manual*.

To set up the link so that it obeys `mouse-1-click-follows-link`, you must either (1) apply a `follow-link` text or overlay property to the link text, or (2) bind the `follow-link` event to a keymap (which can be a major mode keymap or a local keymap specified via the `keymap` text property). The value of the `follow-link` property, or the binding for the `follow-link` event, acts as a condition for the link action. This condition tells Emacs two things: the circumstances under which a *`mouse-1`* click should be regarded as occurring inside the link, and how to compute an action code that says what to translate the *`mouse-1`* click into. The link action condition can be one of the following:

`mouse-face`
: If the condition is the symbol `mouse-face`, a position is inside a link if there is a non-`nil` `mouse-face` property at that position. The action code is always `t`.

  For example, here is how Info mode handles `mouse-1`:

  ```
  (keymap-set Info-mode-map "<follow-link>" 'mouse-face)
  ```

a function
: If the condition is a function, *func*, then a position *pos* is inside a link if `(`*`func pos`*`)` evaluates to non-`nil`. The value returned by *func* serves as the action code.

  For example, here is how pcvs enables *`mouse-1`* to follow links on file names only:

  ```
  (keymap-set map "<follow-link>"
              (lambda (pos)
                (eq (get-char-property pos 'face) 'cvs-filename-face)))
  ```

anything else
: If the condition value is anything else, then the position is inside a link and the condition itself is the action code. Clearly, you should specify this kind of condition only when applying the condition via a text or overlay property on the link text (so that it does not apply to the entire buffer).

The action code tells *`mouse-1`* how to follow the link:

a string or vector
: If the action code is a string or vector, the *`mouse-1`* event is translated into the first element of the string or vector; i.e., the action of the *`mouse-1`* click is the

local or global binding of that character or symbol. Thus, if the action code is `"foo"`, *`mouse-1`* translates into *`f`*. If it is `[foo]`, *`mouse-1`* translates into `foo`.

anything else
: For any other non-`nil` action code, the *`mouse-1`* event is translated into a *`mouse-2`* event at the same position.

To define *`mouse-1`* to activate a button defined with `define-button-type`, give the button a `follow-link` property. The property value should be a link action condition, as described above. See Section 41.20 [Buttons], page 1236. For example, here is how Help mode handles *`mouse-1`*:

```
(define-button-type 'help-xref
  'follow-link t
  'action #'help-button-action)
```

To define *`mouse-1`* on a widget defined with `define-widget`, give the widget a `:follow-link` property. The property value should be a link action condition, as described above. For example, here is how the `link` widget specifies that a `mouse-1` click shall be translated to `RET`:

```
(define-widget 'link 'item
  "An embedded link."
  :button-prefix 'widget-link-prefix
  :button-suffix 'widget-link-suffix
  :follow-link "\C-m"
  :help-echo "Follow the link."
  :format "%[%t%]")
```

`mouse-on-link-p` *pos* [Function]
: This function returns non-`nil` if position *pos* in the current buffer is on a link. *pos* can also be a mouse event location, as returned by `event-start` (see Section 22.7.15 [Accessing Mouse], page 457).

33.19.9 Defining and Using Fields

A field is a range of consecutive characters in the buffer that are identified by having the same value (comparing with `eq`) of the `field` property (either a text-property or an overlay property). This section describes special functions that are available for operating on fields.

You specify a field with a buffer position, *pos*. We think of each field as containing a range of buffer positions, so the position you specify stands for the field containing that position.

When the characters before and after *pos* are part of the same field, there is no doubt which field contains *pos*: the one those characters both belong to. When *pos* is at a boundary between fields, which field it belongs to depends on the stickiness of the `field` properties of the two surrounding characters (see Section 33.19.6 [Sticky Properties], page 935). The field whose property would be inherited by text inserted at *pos* is the field that contains *pos*.

There is an anomalous case where newly inserted text at *pos* would not inherit the `field` property from either side. This happens if the previous character's `field` property is not rear-sticky, and the following character's `field` property is not front-sticky. In this case, *pos* belongs to neither the preceding field nor the following field; the field functions treat it as belonging to an empty field whose beginning and end are both at *pos*.

In all of these functions, if *pos* is omitted or `nil`, the value of point is used by default. If narrowing is in effect, then *pos* should fall within the accessible portion. See Section 31.4 [Narrowing], page 867.

`field-beginning` **&optional** *pos escape-from-edge limit* [Function]
This function returns the beginning of the field specified by *pos*.

If *pos* is at the beginning of its field, and *escape-from-edge* is non-`nil`, then the return value is always the beginning of the preceding field that *ends* at *pos*, regardless of the stickiness of the `field` properties around *pos*.

If *limit* is non-`nil`, it is a buffer position; if the beginning of the field is before *limit*, then *limit* will be returned instead.

`field-end` **&optional** *pos escape-from-edge limit* [Function]
This function returns the end of the field specified by *pos*.

If *pos* is at the end of its field, and *escape-from-edge* is non-`nil`, then the return value is always the end of the following field that *begins* at *pos*, regardless of the stickiness of the `field` properties around *pos*.

If *limit* is non-`nil`, it is a buffer position; if the end of the field is after *limit*, then *limit* will be returned instead.

`field-string` **&optional** *pos* [Function]
This function returns the contents of the field specified by *pos*, as a string.

`field-string-no-properties` **&optional** *pos* [Function]
This function returns the contents of the field specified by *pos*, as a string, discarding text properties.

`delete-field` **&optional** *pos* [Function]
This function deletes the text of the field specified by *pos*.

`constrain-to-field` *new-pos old-pos* **&optional** *escape-from-edge only-in-line inhibit-capture-property* [Function]
This function constrains *new-pos* to the field that *old-pos* belongs to—in other words, it returns the position closest to *new-pos* that is in the same field as *old-pos*.

If *new-pos* is `nil`, then `constrain-to-field` uses the value of point instead, and moves point to the resulting position in addition to returning that position.

If *old-pos* is at the boundary of two fields, then the acceptable final positions depend on the argument *escape-from-edge*. If *escape-from-edge* is `nil`, then *new-pos* must be in the field whose `field` property equals what new characters inserted at *old-pos* would inherit. (This depends on the stickiness of the `field` property for the characters before and after *old-pos*.) If *escape-from-edge* is non-`nil`, *new-pos* can be anywhere in the two adjacent fields. Additionally, if two fields are separated by another field with the special value `boundary`, then any point within this special field is also considered to be on the boundary.

Commands like `C-a` with no argument, that normally move backward to a specific kind of location and stay there once there, probably should specify `nil` for *escape-from-edge*. Other motion commands that check fields should probably pass `t`.

If the optional argument *only-in-line* is non-`nil`, and constraining *new-pos* in the usual way would move it to a different line, *new-pos* is returned unconstrained. This used in commands that move by line, such as `next-line` and `beginning-of-line`, so that they respect field boundaries only in the case where they can still move to the right line.

If the optional argument *inhibit-capture-property* is non-`nil`, and *old-pos* has a non-`nil` property of that name, then any field boundaries are ignored.

You can cause `constrain-to-field` to ignore all field boundaries (and so never constrain anything) by binding the variable `inhibit-field-text-motion` to a non-`nil` value.

33.19.10 Why Text Properties are not Intervals

Some editors that support adding attributes to text in the buffer do so by letting the user specify intervals within the text, and adding the properties to the intervals. Those editors permit the user or the programmer to determine where individual intervals start and end. We deliberately provided a different sort of interface in Emacs Lisp to avoid certain paradoxical behavior associated with text modification.

If the actual subdivision into intervals is meaningful, that means you can distinguish between a buffer that is just one interval with a certain property, and a buffer containing the same text subdivided into two intervals, both of which have that property.

Suppose you take the buffer with just one interval and kill part of the text. The text remaining in the buffer is one interval, and the copy in the kill ring (and the undo list) becomes a separate interval. Then if you yank back the killed text, you get two intervals with the same properties. Thus, editing does not preserve the distinction between one interval and two.

Suppose we attempt to fix this problem by coalescing the two intervals when the text is inserted. That works fine if the buffer originally was a single interval. But suppose instead that we have two adjacent intervals with the same properties, and we kill the text of one interval and yank it back. The same interval-coalescence feature that rescues the other case causes trouble in this one: after yanking, we have just one interval. Once again, editing does not preserve the distinction between one interval and two.

Insertion of text at the border between intervals also raises questions that have no satisfactory answer.

However, it is easy to arrange for editing to behave consistently for questions of the form, "What are the properties of text at this buffer or string position?" So we have decided these are the only questions that make sense; we have not implemented asking questions about where intervals start or end.

In practice, you can usually use the text property search functions in place of explicit interval boundaries. You can think of them as finding the boundaries of intervals, assuming that intervals are always coalesced whenever possible. See Section 33.19.3 [Property Search], page 924.

Emacs also provides explicit intervals as a presentation feature; see Section 41.9 [Overlays], page 1154.

33.20 Substituting for a Character Code

The following functions replace characters within a specified region based on their character codes.

`subst-char-in-region` *start end old-char new-char* **&optional** *noundo* [Function]

This function replaces all occurrences of the character *old-char* with the character *new-char* in the region of the current buffer defined by *start* and *end*. Both characters must have the same length of their multibyte form.

If *noundo* is non-`nil`, then `subst-char-in-region` does not record the change for undo and does not mark the buffer as modified. This was useful for controlling the old selective display feature (see Section 41.7 [Selective Display], page 1149).

`subst-char-in-region` does not move point and returns `nil`.

```
---------- Buffer: foo ----------
This is the contents of the buffer before.
---------- Buffer: foo ----------

(subst-char-in-region 1 20 ?i ?X)
     ⇒ nil

---------- Buffer: foo ----------
ThXs Xs the contents of the buffer before.
---------- Buffer: foo ----------
```

`subst-char-in-string` *fromchar tochar string* **&optional** *inplace* [Function]

This function replaces all occurrences of the character *fromchar* with *tochar* in *string*. By default, substitution occurs in a copy of *string*, but if the optional argument *inplace* is non-`nil`, the function modifies the *string* itself. In any case, the function returns the resulting string.

`translate-region` *start end table* [Command]

This function applies a translation table to the characters in the buffer between positions *start* and *end*.

The translation table *table* is a string or a char-table; `(aref table ochar)` gives the translated character corresponding to *ochar*. If *table* is a string, any characters with codes larger than the length of *table* are not altered by the translation.

The return value of `translate-region` is the number of characters that were actually changed by the translation. This does not count characters that were mapped into themselves in the translation table.

33.21 Registers

A register is a sort of variable used in Emacs editing that can hold a variety of different kinds of values. Each register is named by a single character. All ASCII characters and their meta variants (but with the exception of `C-g`) can be used to name registers. Thus, there are 255 possible registers. A register is designated in Emacs Lisp by the character that is its name.

`register-alist` [Variable]
: This variable is an alist of elements of the form `(`*`name`* `.` *`contents`*`)`. Normally, there is one element for each Emacs register that has been used.

  The object *name* is a character (an integer) identifying the register.

The *contents* of a register can have several possible types:

a number
: A number stands for itself. If `insert-register` finds a number in the register, it converts the number to decimal.

a marker
: A marker represents a buffer position to jump to.

a string
: A string is text saved in the register.

a rectangle
: A rectangle is represented by a list of strings.

`(`*`window-configuration position`*`)`
: This represents a window configuration to restore in one frame, and a position to jump to in the current buffer.

`(`*`frame-configuration position`*`)`
: This represents a frame configuration to restore, and a position to jump to in the current buffer. Frame configurations are also known as *framesets*.

`(file` *`filename`*`)`
: This represents a file to visit; jumping to this value visits file *filename*.

`(file-query` *`filename position`*`)`
: This represents a file to visit and a position in it; jumping to this value visits file *filename* and goes to buffer position *position*. Restoring this type of position asks the user for confirmation first.

`(buffer` *`buffer-name`*`)`
: This represents a buffer; jumping to this value switches to buffer *buffer-name*.

The functions in this section return unpredictable values unless otherwise stated.

`get-register` *reg* [Function]
: This function returns the contents of the register *reg*, or `nil` if it has no contents.

`set-register` *reg value* [Function]
: This function sets the contents of register *reg* to *value*. A register can be set to any value, but the other register functions expect only certain data types. The return value is *value*.

`view-register` *reg* [Command]
: This command displays what is contained in register *reg*.

`insert-register` *reg* **&optional** *beforep* [Command]
: This command inserts contents of register *reg* into the current buffer.

  Normally, this command puts point before the inserted text, and the mark after it. However, if the optional second argument *beforep* is non-`nil`, it puts the mark before and point after.

When called interactively, the command defaults to putting point after text, and a prefix argument inverts this behavior.

If the register contains a rectangle, then the rectangle is inserted with its upper left corner at point. This means that text is inserted in the current line and underneath it on successive lines.

If the register contains something other than saved text (a string) or a rectangle (a list), currently useless things happen. This may be changed in the future.

`register-read-with-preview` *prompt* [Function]

This function reads and returns a register name, prompting with *prompt* and possibly showing a preview of the existing registers and their contents. The preview is shown in a temporary window, after the delay specified by the user option `register-preview-delay`, if its value and `register-alist` are both non-`nil`. The preview is also shown if the user requests help (e.g., by typing the help character). We recommend that all interactive commands which read register names use this function.

33.22 Transposition of Text

This function can be used to transpose stretches of text:

`transpose-regions` *start1 end1 start2 end2* **&optional** *leave-markers* [Function]

This function exchanges two nonoverlapping portions of the buffer (if they overlap, the function signals an error). Arguments *start1* and *end1* specify the bounds of one portion and arguments *start2* and *end2* specify the bounds of the other portion.

Normally, `transpose-regions` relocates markers with the transposed text; a marker previously positioned within one of the two transposed portions moves along with that portion, thus remaining between the same two characters in their new position. However, if *leave-markers* is non-`nil`, `transpose-regions` does not do this—it leaves all markers unrelocated.

33.23 Replacing Buffer Text

You can use the following function to replace the text of one buffer with the text of another buffer:

`replace-buffer-contents` *source* **&optional** *max-secs max-costs* [Command]

This function replaces the accessible portion of the current buffer with the accessible portion of the buffer *source*. *source* may either be a buffer object or the name of a buffer. When `replace-buffer-contents` succeeds, the text of the accessible portion of the current buffer will be equal to the text of the accessible portion of the *source* buffer.

This function attempts to keep point, markers, text properties, and overlays in the current buffer intact. One potential case where this behavior is useful is external code formatting programs: they typically write the reformatted text into a temporary buffer or file, and using `delete-region` and `insert-buffer-substring` would destroy these properties. However, the latter combination is typically faster (See Section 33.6 [Deletion], page 888, and Section 33.4 [Insertion], page 884).

For its working, `replace-buffer-contents` needs to compare the contents of the original buffer with that of *source* which is a costly operation if the buffers are huge and there is a high number of differences between them. In order to keep `replace-buffer-contents`'s runtime in bounds, it has two optional arguments.

max-secs defines a hard boundary in terms of seconds. If given and exceeded, it will fall back to `delete-region` and `insert-buffer-substring`.

max-costs defines the quality of the difference computation. If the actual costs exceed this limit, heuristics are used to provide a faster but suboptimal solution. The default value is 1000000.

`replace-buffer-contents` returns `t` if a non-destructive replacement could be performed. Otherwise, i.e., if *max-secs* was exceeded, it returns `nil`.

`replace-region-contents` *beg end replace-fn* **&optional** *max-secs max-costs* [Function]

This function replaces the region between *beg* and *end* using the given *replace-fn*. The function *replace-fn* is run in the current buffer narrowed to the specified region and it should return either a string or a buffer replacing the region.

The replacement is performed using `replace-buffer-contents` (see above) which also describes the *max-secs* and *max-costs* arguments and the return value.

Note: If the replacement is a string, it will be placed in a temporary buffer so that `replace-buffer-contents` can operate on it. Therefore, if you already have the replacement in a buffer, it makes no sense to convert it to a string using `buffer-substring` or similar.

33.24 Dealing With Compressed Data

When `auto-compression-mode` is enabled, Emacs automatically uncompresses compressed files when you visit them, and automatically recompresses them if you alter and save them. See Section "Compressed Files" in *The GNU Emacs Manual*.

The above feature works by calling an external executable (e.g., `gzip`). Emacs can also be compiled with support for built-in decompression using the zlib library, which is faster than calling an external program.

`zlib-available-p` [Function]

This function returns non-`nil` if built-in zlib decompression is available.

`zlib-decompress-region` *start end* **&optional** *allow-partial* [Function]

This function decompresses the region between *start* and *end*, using built-in zlib decompression. The region should contain data that were compressed with gzip or zlib. On success, the function replaces the contents of the region with the decompressed data. If *allow-partial* is `nil` or omitted, then on failure, the function leaves the region unchanged and returns `nil`. Otherwise, it returns the number of bytes that were not decompressed and replaces the region text by whatever data was successfully decompressed. This function can be called only in unibyte buffers.

33.25 Base 64 Encoding

Base 64 code is used in email to encode a sequence of 8-bit bytes as a longer sequence of ASCII graphic characters. It is defined in Internet RFC[1]2045 and also in RFC 4648. This section describes the functions for converting to and from this code.

`base64-encode-region` *beg end* **&optional** *no-line-break* [Command]
This function converts the region from *beg* to *end* into base 64 code. It returns the length of the encoded text. An error is signaled if a character in the region is multibyte, i.e., in a multibyte buffer the region must contain only ASCII characters or raw bytes.

Normally, this function inserts newline characters into the encoded text, to avoid overlong lines. However, if the optional argument *no-line-break* is non-`nil`, these newlines are not added, so the output is just one long line.

`base64url-encode-region` *beg end* **&optional** *no-pad* [Command]
This function is like `base64-encode-region`, but it implements the URL variant of base 64 encoding, per RFC 4648, and it doesn't insert newline characters into the encoded text, so the output is just one long line.

If the optional argument *no-pad* is non-`nil` then this function doesn't generate the padding (`=`).

`base64-encode-string` *string* **&optional** *no-line-break* [Function]
This function converts the string *string* into base 64 code. It returns a string containing the encoded text. As for `base64-encode-region`, an error is signaled if a character in the string is multibyte.

Normally, this function inserts newline characters into the encoded text, to avoid overlong lines. However, if the optional argument *no-line-break* is non-`nil`, these newlines are not added, so the result string is just one long line.

`base64url-encode-string` *string* **&optional** *no-pad* [Function]
Like `base64-encode-string`, but generates the URL variant of base 64, and doesn't insert newline characters into the encoded text, so the result is just one long line.

If the optional argument *no-pad* is non-`nil` then this function doesn't generate the padding.

`base64-decode-region` *beg end* **&optional** *base64url ignore-invalid* [Command]
This function converts the region from *beg* to *end* from base 64 code into the corresponding decoded text. It returns the length of the decoded text.

The decoding functions ignore newline characters in the encoded text.

If optional argument *base64url* is non-`nil`, then padding is optional, and the URL variant of base 64 encoding is used. If optional argument *ignore-invalid* is non-`nil`, then any unrecognized characters are ignored.

[1] An RFC, an acronym for *Request for Comments*, is a numbered Internet informational document describing a standard. RFCs are usually written by technical experts acting on their own initiative, and are traditionally written in a pragmatic, experience-driven manner.

`base64-decode-string` *string* **&optional** *base64url ignore-invalid* [Function]
This function converts the string *string* from base 64 code into the corresponding decoded text. It returns a unibyte string containing the decoded text.

The decoding functions ignore newline characters in the encoded text.

If optional argument *base64url* is non-`nil`, then padding is optional, and the URL variant of base 64 encoding is used. If optional argument *ignore-invalid* is non-`nil`, then any unrecognized characters are ignored.

33.26 Checksum/Hash

Emacs has built-in support for computing *cryptographic hashes*. A cryptographic hash, or *checksum*, is a digital fingerprint of a piece of data (e.g., a block of text) which can be used to check that you have an unaltered copy of that data.

Emacs supports several common cryptographic hash algorithms: MD5, SHA-1, SHA-2, SHA-224, SHA-256, SHA-384 and SHA-512. MD5 is the oldest of these algorithms, and is commonly used in *message digests* to check the integrity of messages transmitted over a network. MD5 and SHA-1 are not collision resistant (i.e., it is possible to deliberately design different pieces of data which have the same MD5 or SHA-1 hash), so you should not use them for anything security-related. For security-related applications you should use the other hash types, such as SHA-2 (e.g. `sha256` or `sha512`).

`secure-hash-algorithms` [Function]
This function returns a list of symbols representing algorithms that `secure-hash` can use.

`secure-hash` *algorithm object* **&optional** *start end binary* [Function]
This function returns a hash for *object*. The argument *algorithm* is a symbol stating which hash to compute: one of `md5`, `sha1`, `sha224`, `sha256`, `sha384` or `sha512`. The argument *object* should be a buffer or a string.

The optional arguments *start* and *end* are character positions specifying the portion of *object* to compute the message digest for. If they are `nil` or omitted, the hash is computed for the whole of *object*.

If the argument *binary* is omitted or `nil`, the function returns the *text form* of the hash, as an ordinary Lisp string. If *binary* is non-`nil`, it returns the hash in *binary form*, as a sequence of bytes stored in a unibyte string. The length of the returned string depends on *algorithm*:

- For `md5`: 32 characters (32 bytes if *binary* is non-`nil`).
- For `sha1`: 40 characters (40 bytes if *binary* is non-`nil`).
- For `sha224`: 56 characters (56 bytes if *binary* is non-`nil`).
- For `sha256`: 64 characters (64 bytes if *binary* is non-`nil`).
- For `sha384`: 96 characters (96 bytes if *binary* is non-`nil`).
- For `sha512`: 128 characters (128 bytes if *binary* is non-`nil`).

This function does not compute the hash directly from the internal representation of *object*'s text (see Section 34.1 [Text Representations], page 966). Instead, it encodes the text using a coding system (see Section 34.10 [Coding Systems], page 980), and

computes the hash from that encoded text. If *object* is a buffer, the coding system used is the one which would be chosen by default for writing the text of that buffer into a file. If *object* is a string, the user's preferred coding system is used (see Section "Recognize Coding" in *GNU Emacs Manual*).

`md5` *object* **&optional** *start end coding-system noerror* [Function]
This function returns an MD5 hash. It is semi-obsolete, since for most purposes it is equivalent to calling `secure-hash` with `md5` as the *algorithm* argument. The *object*, *start* and *end* arguments have the same meanings as in `secure-hash`. The function returns a 32-character string.

If *coding-system* is non-`nil`, it specifies a coding system to use to encode the text; if omitted or `nil`, the default coding system is used, like in `secure-hash`.

Normally, `md5` signals an error if the text can't be encoded using the specified or chosen coding system. However, if *noerror* is non-`nil`, it silently uses `raw-text` coding instead.

`buffer-hash` **&optional** *buffer-or-name* [Function]
Return a hash of *buffer-or-name*. If `nil`, this defaults to the current buffer. As opposed to `secure-hash`, this function computes the hash based on the internal representation of the buffer, disregarding any coding systems. It's therefore only useful when comparing two buffers running in the same Emacs, and is not guaranteed to return the same hash between different Emacs versions. It should be somewhat more efficient on larger buffers than `secure-hash` is, and should not allocate more memory.

`sha1` *object* **&optional** *start end binary* [Function]
This function is equivalent to calling `secure-hash` like this:

```
(secure-hash 'sha1 object start end binary)
```

It returns a 40-character string if *binary* is `nil`, or a 40-byte unibyte string otherwise.

33.27 Suspicious Text

Emacs can display text from many external sources, like email and Web sites. Attackers may attempt to confuse the user reading this text by using obfuscated URLs or email addresses, and tricking the user into visiting a web page they didn't intend to visit, or sending an email to the wrong address.

This usually involves using characters from scripts that visually look like ASCII characters (i.e., are homoglyphs), but there are also other techniques used, like using bidirectional overrides, or having an HTML link text that says one thing, while the underlying URL points somewhere else.

To help identify these *suspicious text strings*, Emacs provides a library to do a number of checks on text. (See UTS #39: Unicode Security Mechanisms (`https://www.unicode.org/reports/tr39/`) for the rationale behind the checks that are available and more details about them.) Packages that present data that might be suspicious should use this library to flag suspicious text on display.

`textsec-suspicious-p` *object type* [Function]

This function is the high-level interface function that packages should use. It respects the `textsec-check` user option, which allows the user to disable the checks.

This function checks *object* (whose data type depends on *type*) to see if it looks suspicious when interpreted as a thing of *type*. The available types and the corresponding *object* data types are:

`domain` Check whether a domain (e.g., '`www.gnu.org`' looks suspicious. *object* should be a string, the domain name.

`url` Check whether an URL (e.g., '`http://gnu.org/foo/bar`') looks suspicious. *object* should be a string, the URL to check.

`link` Check whether an HTML link (e.g., '`<a href='http://gnu.org'>fsf.org</a>`' looks suspicious. In this case, *object* should be a `cons` cell where the `car` is the URL string, and the `cdr` is the link text. The link is deemed suspicious if the link text contains a domain name, and that domain name points to something other than the URL.

`email-address`
Check whether an email address (e.g., '`foo@example.org`') looks suspicious. *object* should be a string.

`local-address`
Check whether the local part of an email address (the bit before the '`@`' sign) looks suspicious. *object* should be a string.

`name` Check whether a name (used in an email address header) looks suspicious. *object* should be a string.

`email-address-header`
Check whether a full RFC2822 email address header (e.g., '`=?utf-8?Q?=C3=81?= <foo@example.com>`') looks suspicious. *object* should be a string.

If *object* is suspicious, this function returns a string that explains why it is suspicious. If *object* is not suspicious, the function returns `nil`.

If the text is suspicious, the application should mark the suspicious text with the `textsec-suspicious` face, and make the explanation returned by `textsec-suspicious-p` available to the user in some way (for example, in a tooltip). The application might also prompt the user for confirmation before taking any action on a suspicious string (like sending an email to a suspicious email address).

33.28 GnuTLS Cryptography

If compiled with GnuTLS, Emacs offers built-in cryptographic support. Following the GnuTLS API terminology, the available tools are digests, MACs, symmetric ciphers, and AEAD ciphers.

The terms used herein, such as IV (Initialization Vector), require some familiarity with cryptography and will not be defined in detail. Please consult `https://www.gnutls.org/` for specific documentation which may help you understand the terminology and structure of the GnuTLS library.

33.28.1 Format of GnuTLS Cryptography Inputs

The inputs to GnuTLS cryptographic functions can be specified in several ways, both as primitive Emacs Lisp types or as lists.

The list form is currently similar to how `md5` and `secure-hash` operate.

`buffer` Simply passing a buffer as input means the whole buffer should be used.

`string` A string as input will be used directly. It may be modified by the function (unlike most other Emacs Lisp functions) to reduce the chance of exposing sensitive data after the function does its work.

(*`buffer-or-string start end coding-system noerror`*)
: This specifies a buffer or a string as described above, but an optional range can be specified with *start* and *end*.

  In addition an optional *coding-system* can be specified if needed.

  The last optional item, *noerror*, overrides the normal error when the text can't be encoded using the specified or chosen coding system. When *noerror* is non-`nil`, this function silently uses `raw-text` coding instead.

(`iv-auto` *`length`*)
: This generates a random IV (Initialization Vector) of the specified length and passes it to the function. This ensures that the IV is unpredictable and unlikely to be reused in the same session.

33.28.2 GnuTLS Cryptographic Functions

`gnutls-digests` [Function]
: This function returns the alist of the GnuTLS digest algorithms.

  Each entry has a key which represents the algorithm, followed by a plist with internal details about the algorithm. The plist will have `:type gnutls-digest-algorithm` and also will have the key `:digest-algorithm-length 64` to indicate the size, in bytes, of the resulting digest.

  There is a name parallel between GnuTLS MAC and digest algorithms but they are separate things internally and should not be mixed.

`gnutls-hash-digest` *digest-method input* [Function]
: The *digest-method* can be the whole plist from `gnutls-digests`, or just the symbol key, or a string with the name of that symbol.

  The *input* can be specified as a buffer or string or in other ways (see Section 33.28.1 [Format of GnuTLS Cryptography Inputs], page 950).

  This function returns `nil` on error, and signals a Lisp error if the *digest-method* or *input* are invalid. On success, it returns a list of a binary string (the output) and the IV used.

`gnutls-macs` [Function]
: This function returns the alist of the GnuTLS MAC algorithms.

  Each entry has a key which represents the algorithm, followed by a plist with internal details about the algorithm. The plist will have `:type gnutls-mac-algorithm`

and also will have the keys `:mac-algorithm-length` `:mac-algorithm-keysize` `:mac-algorithm-noncesize` to indicate the size, in bytes, of the resulting hash, the key, and the nonce respectively.

The nonce is currently unused and only some MACs support it.

There is a name parallel between GnuTLS MAC and digest algorithms but they are separate things internally and should not be mixed.

`gnutls-hash-mac` *hash-method key input* [Function]

The *hash-method* can be the whole plist from `gnutls-macs`, or just the symbol key, or a string with the name of that symbol.

The *key* can be specified as a buffer or string or in other ways (see Section 33.28.1 [Format of GnuTLS Cryptography Inputs], page 950). The *key* will be wiped after use if it's a string.

The *input* can be specified as a buffer or string or in other ways (see Section 33.28.1 [Format of GnuTLS Cryptography Inputs], page 950).

This function returns `nil` on error, and signals a Lisp error if the *hash-method* or *key* or *input* are invalid.

On success, it returns a list of a binary string (the output) and the IV used.

`gnutls-ciphers` [Function]

This function returns the alist of the GnuTLS ciphers.

Each entry has a key which represents the cipher, followed by a plist with internal details about the algorithm. The plist will have `:type` `gnutls-symmetric-cipher` and also will have the keys `:cipher-aead-capable` set to `nil` or `t` to indicate AEAD capability; and `:cipher-tagsize` `:cipher-blocksize` `:cipher-keysize` `:cipher-ivsize` to indicate the size, in bytes, of the tag, block size of the resulting data, the key, and the IV respectively.

`gnutls-symmetric-encrypt` *cipher key iv input* **&optional** *aead_auth* [Function]

The *cipher* can be the whole plist from `gnutls-ciphers`, or just the symbol key, or a string with the name of that symbol.

The *key* can be specified as a buffer or string or in other ways (see Section 33.28.1 [Format of GnuTLS Cryptography Inputs], page 950). The *key* will be wiped after use if it's a string.

The *iv* and *input* and the optional *aead_auth* can be specified as a buffer or string or in other ways (see Section 33.28.1 [Format of GnuTLS Cryptography Inputs], page 950).

aead_auth is only checked with AEAD ciphers, that is, ciphers whose plist has `:cipher-aead-capable` `t`. Otherwise it's ignored.

This function returns `nil` on error, and signals a Lisp error if the *cipher* or *key*, *iv*, or *input* are invalid, or if *aead_auth* was specified with an AEAD cipher and was invalid.

On success, it returns a list of a binary string (the output) and the IV used.

`gnutls-symmetric-decrypt` *cipher key iv input* **&optional** *aead_auth* [Function]

The *cipher* can be the whole plist from `gnutls-ciphers`, or just the symbol key, or a string with the name of that symbol.

The *key* can be specified as a buffer or string or in other ways (see Section 33.28.1 [Format of GnuTLS Cryptography Inputs], page 950). The *key* will be wiped after use if it's a string.

The *iv* and *input* and the optional *aead_auth* can be specified as a buffer or string or in other ways (see Section 33.28.1 [Format of GnuTLS Cryptography Inputs], page 950).

aead_auth is only checked with AEAD ciphers, that is, ciphers whose plist has `:cipher-aead-capable t`. Otherwise it's ignored.

This function returns `nil` on decryption error, and signals a Lisp error if the *cipher* or *key*, *iv*, or *input* are invalid, or if *aead_auth* was specified with an AEAD cipher and was invalid.

On success, it returns a list of a binary string (the output) and the IV used.

33.29 Database

Emacs can be compiled with built-in support for accessing SQLite databases. This section describes the facilities available for accessing SQLite databases from Lisp programs.

`sqlite-available-p` [Function]
The function returns non-`nil` if built-in SQLite support is available in this Emacs session.

When SQLite support is available, the following functions can be used.

`sqlite-open` **&optional** *file* [Function]
This function opens *file* as an SQLite database file. If *file* doesn't exist, a new database will be created and stored in that file. If *file* is omitted or `nil`, a new in-memory database is created instead.

The return value is a *database object* that can be used as the argument to most of the subsequent functions described below.

`sqlitep` *object* [Function]
This predicate returns non-`nil` if *object* is an SQLite database object. The database object returned by the `sqlite-open` function satisfies this predicate.

`sqlite-close` *db* [Function]
Close the database *db*. It's usually not necessary to call this function explicitly—the database will automatically be closed if Emacs shuts down or the database object is garbage collected.

`sqlite-execute` *db statement* **&optional** *values* [Function]
Execute the SQL *statement*. For instance:

```
(sqlite-execute db "insert into foo values ('bar', 2)")
```

If the optional *values* parameter is present, it should be either a list or a vector of values to bind while executing the statement. For instance:

```
(sqlite-execute db "insert into foo values (?, ?)" '("bar" 2))
```

This has exactly the same effect as the previous example, but is more efficient and safer (because it doesn't involve any string parsing or interpolation).

`sqlite-execute` usually returns the number of affected rows. For instance, an '`insert`' statement will typically return '`1`', whereas an '`update`' statement may return zero or a higher number. However, when using SQL statements like '`insert into ... returning ...`' and the like, the values specified by '`returning ...`' will be returned instead.

Strings in SQLite are, by default, stored as `utf-8`, and selecting a text column will decode the string using that charset. Selecting a blob column will return the raw data without any decoding (i.e., it will return a unibyte string containing the bytes as stored in the database). Inserting binary data into blob columns, however, requires some care, as `sqlite-execute` will, by default, interpret all strings as `utf-8`.

So if you have, for instance, GIF data in a unibyte string called *gif*, you have to mark it specially to let `sqlite-execute` know this:

```
(put-text-property 0 1 'coding-system 'binary gif)
(sqlite-execute db "insert into foo values (?, ?)" (list gif 2))
```

`sqlite-select` *db query* **&optional** *values return-type* [Function]

Select some data from *db* and return them. For instance:

```
(sqlite-select db "select * from foo where key = 2")
  ⇒ (("bar" 2))
```

As with the `sqlite-execute`, you can optionally pass in a list or a vector of values that will be bound before executing the select:

```
(sqlite-select db "select * from foo where key = ?" [2])
  ⇒ (("bar" 2))
```

This is usually more efficient and safer than the method used by the previous example.

By default, this function returns a list of matching rows, where each row is a list of column values. If *return-type* is `full`, the names of the columns (as a list of strings) will be returned as the first element in the return value.

If *return-type* is `set`, this function will return a *statement object* instead. This object can be examined by using the `sqlite-next`, `sqlite-columns` and `sqlite-more-p` functions. If the result set is small, it's often more convenient to just return the data directly, but if the result set is large (or if you won't be using all the data from the set), using the `set` method will allocate a lot less memory, and is therefore more memory-efficient.

`sqlite-next` *statement* [Function]

This function returns the next row in the result set *statement*, typically an object returned by `sqlite-select`.

```
(sqlite-next stmt)
    ⇒ ("bar" 2)
```

`sqlite-columns` *statement* [Function]

This function returns the column names of the result set *statement*, typically an object returned by `sqlite-select`.

```
(sqlite-columns stmt)
    ⇒ ("name" "issue")
```

`sqlite-more-p` *statement* [Function]
: This predicate says whether there is more data to be fetched from the result set *statement*, typically an object returned by `sqlite-select`.

`sqlite-finalize` *statement* [Function]
: If *statement* is not going to be used any more, calling this function will free the resources used by *statement*. This is usually not necessary—when the *statement* object is garbage-collected, Emacs will automatically free its resources.

`sqlite-transaction` *db* [Function]
: Start a transaction in *db*. When in a transaction, other readers of the database won't access the results until the transaction has been committed by `sqlite-commit`.

`sqlite-commit` *db* [Function]
: End a transaction in *db* and write the data out to its file.

`sqlite-rollback` *db* [Function]
: End a transaction in *db* and discard any changes that have been made by the transaction.

`with-sqlite-transaction` *db body...* [Macro]
: Like `progn` (see Section 11.1 [Sequencing], page 158), but executes *body* with a transaction held, and commits the transaction at the end if *body* completes normally. If *body* signals an error, or committing the transaction fails, the changes in *db* performed by *body* are rolled back. The macro returns the value of *body* if it completes normally and commit succeeds.

`sqlite-pragma` *db pragma* [Function]
: Execute *pragma* in *db*. A *pragma* is usually a command that affects the database overall, instead of any particular table. For instance, to make SQLite automatically garbage collect data that's no longer needed, you can say:

```
(sqlite-pragma db "auto_vacuum = FULL")
```

: This function returns non-`nil` on success and `nil` if the pragma failed. Many pragmas can only be issued when the database is brand new and empty.

`sqlite-load-extension` *db module* [Function]
: Load the named extension *module* into the database *db*. Extensions are usually shared-library files; on GNU and Unix systems, they have the `.so` file-name extension.

`sqlite-version` [Function]
: Return a string denoting the version of the SQLite library in use.

If you wish to list the contents of an SQLite file, you can use the `sqlite-mode-open-file` command. This will pop to a buffer using `sqlite-mode`, which allows you to examine (and alter) the contents of an SQLite database.

33.30 Parsing HTML and XML

Emacs can be compiled with built-in `libxml2` support.

`libxml-available-p` [Function]
: This function returns non-`nil` if built-in libxml2 support is available in this Emacs session.

When libxml2 support is available, the following functions can be used to parse HTML or XML text into Lisp object trees.

`libxml-parse-html-region` **&optional** *start end base-url discard-comments* [Function]
: This function parses the text between *start* and *end* as HTML, and returns a list representing the HTML *parse tree*. It attempts to handle real-world HTML by robustly coping with syntax mistakes.

  If *start* or *end* are `nil`, they default to the values from `point-min` and `point-max`, respectively.

  The optional argument *base-url*, if non-`nil`, should be used for warnings and errors reported by the `libxml2` library, but Emacs currently calls the library with errors and warnings disabled, so this argument is not used.

  If the optional argument *discard-comments* is non-`nil`, any top-level comment is discarded. (This argument is obsolete and will be removed in future Emacs versions. To remove comments, use the `xml-remove-comments` utility function on the data before you call the parsing function.)

  In the parse tree, each HTML node is represented by a list in which the first element is a symbol representing the node name, the second element is an alist of node attributes, and the remaining elements are the subnodes.

  The following example demonstrates this. Given this (malformed) HTML document:

  ```
  <html><head></head><body width=101><div class=thing>Foo<div>Yes
  ```

  A call to `libxml-parse-html-region` returns this DOM (document object model):

  ```
  (html nil
   (head nil)
   (body ((width . "101"))
    (div ((class . "thing"))
     "Foo"
     (div nil
      "Yes"))))
  ```

`shr-insert-document` *dom* [Function]
: This function renders the parsed HTML in *dom* into the current buffer. The argument *dom* should be a list as generated by `libxml-parse-html-region`. This function is, e.g., used by *The Emacs Web Wowser Manual*.

`libxml-parse-xml-region` **&optional** *start end base-url discard-comments* [Function]
: This function is the same as `libxml-parse-html-region`, except that it parses the text as XML rather than HTML (so it is stricter about syntax).

33.30.1 Document Object Model

The DOM returned by `libxml-parse-html-region` (and the other XML parsing functions) is a tree structure where each node has a node name (called a *tag*), and optional key/value *attribute* list, and then a list of *child nodes*. The child nodes are either strings or DOM objects.

```
(body ((width . "101"))
 (div ((class . "thing"))
  "Foo"
  (div nil
   "Yes")))
```

`dom-node` *tag* **&optional** *attributes* **&rest** *children* [Function]
: This function creates a DOM node of type *tag*. If given, *attributes* should be a key/value pair list. If given, *children* should be DOM nodes.

The following functions can be used to work with this structure. Each function takes a DOM node, or a list of nodes. In the latter case, only the first node in the list is used.

Simple accessors:

`dom-tag` *node*
: Return the *tag* (also called "node name") of the node.

`dom-attr` *node attribute*
: Return the value of *attribute* in the node. A common usage would be:

```
(dom-attr img 'href)
=> "https://fsf.org/logo.png"
```

`dom-children` *node*
: Return all the children of the node.

`dom-non-text-children` *node*
: Return all the non-string children of the node.

`dom-attributes` *node*
: Return the key/value pair list of attributes of the node.

`dom-text` *node*
: Return all the textual elements of the node as a concatenated string.

`dom-texts` *node*
: Return all the textual elements of the node, as well as the textual elements of all the children of the node, recursively, as a concatenated string. This function also takes an optional separator to be inserted between the textual elements.

`dom-parent` *dom node*
: Return the parent of *node* in *dom*.

`dom-remove` *dom node*
: Remove *node* from *dom*.

The following are functions for altering the DOM.

`dom-set-attribute` *node attribute value*
: Set the *attribute* of the node to *value*.

`dom-remove-attribute` *`node attribute`*
: Remove *attribute* from *node*.

`dom-append-child` *`node child`*
: Append *child* as the last child of *node*.

`dom-add-child-before` *`node child before`*
: Add *child* to *node*'s child list before the *before* node. If *before* is `nil`, make *child* the first child.

`dom-set-attributes` *`node attributes`*
: Replace all the attributes of the node with a new key/value list.

The following are functions for searching for elements in the DOM. They all return lists of matching nodes.

`dom-by-tag` *`dom tag`*
: Return all nodes in *dom* that are of type *tag*. A typical use would be:

```
(dom-by-tag dom 'td)
=> '((td ...) (td ...) (td ...))
```

`dom-by-class` *`dom match`*
: Return all nodes in *dom* that have class names that match *match*, which is a regular expression.

`dom-by-style` *`dom style`*
: Return all nodes in *dom* that have styles that match *match*, which is a regular expression.

`dom-by-id` *`dom style`*
: Return all nodes in *dom* that have IDs that match *match*, which is a regular expression.

`dom-search` *`dom predicate`*
: Return all nodes in *dom* where *predicate* returns a non-`nil` value. *predicate* is called with the node to be tested as its parameter.

`dom-strings` *`dom`*
: Return all strings in *dom*.

Utility functions:

`dom-pp` *`dom`* `&optional` *`remove-empty`*
: Pretty-print *dom* at point. If *remove-empty*, don't print textual nodes that just contain white-space.

`dom-print` *`dom`* `&optional` *`pretty xml`*
: Print *dom* at point. If *xml* is non-`nil`, print as XML; otherwise, print as HTML. If *pretty* is non-`nil`, indent the HTML/XML logically.

33.31 Parsing and generating JSON values

When Emacs is compiled with JSON (*JavaScript Object Notation*) support, it provides several functions to convert between Lisp objects and JSON values. Any JSON value can be converted to a Lisp object, but not vice versa. Specifically:

- JSON uses three keywords: `true`, `null`, `false`. `true` is represented by the symbol `t`. By default, the remaining two are represented, respectively, by the symbols `:null` and `:false`.
- JSON only has floating-point numbers. They can represent both Lisp integers and Lisp floating-point numbers.
- JSON strings are always Unicode strings encoded in UTF-8. Lisp strings can contain non-Unicode characters.
- JSON has only one sequence type, the array. JSON arrays are represented using Lisp vectors.
- JSON has only one map type, the object. JSON objects are represented using Lisp hashtables, alists or plists. When an alist or plist contains several elements with the same key, Emacs uses only the first element for serialization, in accordance with the behavior of `assq`.

Note that `nil`, being both a valid alist and a valid plist, represents `{}`, the empty JSON object; not `null`, `false`, or an empty array, all of which are different JSON values.

`json-available-p` [Function]
: This predicate returns non-`nil` if Emacs has been built with JSON support, and the library is available on the current system.

If some Lisp object can't be represented in JSON, the serialization functions will signal an error of type `wrong-type-argument`. The parsing functions can also signal the following errors:

`json-unavailable`
: Signaled when the parsing library isn't available.

`json-end-of-file`
: Signaled when encountering a premature end of the input text.

`json-trailing-content`
: Signaled when encountering unexpected input after the first JSON object parsed.

`json-parse-error`
: Signaled when encountering invalid JSON syntax.

Top-level values and the subobjects within these top-level values can be serialized to JSON. Likewise, the parsing functions will return any of the possible types described above.

`json-serialize` *object* **&rest** *args* [Function]
: This function returns a new Lisp string which contains the JSON representation of *object*. The argument *args* is a list of keyword/argument pairs. The following keywords are accepted:

`:null-object`
: The value decides which Lisp object to use to represent the JSON keyword `null`. It defaults to the symbol `:null`.

`:false-object`
: The value decides which Lisp object to use to represent the JSON keyword `false`. It defaults to the symbol `:false`.

`json-insert` *object* **&rest** *args* [Function]

This function inserts the JSON representation of *object* into the current buffer before point. The argument *args* are interpreted as in `json-parse-string`.

`json-parse-string` *string* **&rest** *args* [Function]

This function parses the JSON value in *string*, which must be a Lisp string. If *string* doesn't contain a valid JSON object, this function signals the `json-parse-error` error.

The argument *args* is a list of keyword/argument pairs. The following keywords are accepted:

`:object-type`
: The value decides which Lisp object to use for representing the key-value mappings of a JSON object. It can be either `hash-table`, the default, to make hashtables with strings as keys; `alist` to use alists with symbols as keys; or `plist` to use plists with keyword symbols as keys.

`:array-type`
: The value decides which Lisp object to use for representing a JSON array. It can be either `array`, the default, to use Lisp arrays; or `list` to use lists.

`:null-object`
: The value decides which Lisp object to use to represent the JSON keyword `null`. It defaults to the symbol `:null`.

`:false-object`
: The value decides which Lisp object to use to represent the JSON keyword `false`. It defaults to the symbol `:false`.

`json-parse-buffer` **&rest** *args* [Function]

This function reads the next JSON value from the current buffer, starting at point. It moves point to the position immediately after the value if contains a valid JSON object; otherwise it signals the `json-parse-error` error and doesn't move point. The arguments *args* are interpreted as in `json-parse-string`.

33.32 JSONRPC communication

The `jsonrpc` library implements the JSONRPC specification, version 2.0, as it is described in `https://www.jsonrpc.org/`. As the name suggests, JSONRPC is a generic *Remote Procedure Call* protocol designed around JSON objects, which you can convert to and from Lisp objects (see Section 33.31 [Parsing JSON], page 958).

33.32.1 Overview

Quoting from the spec (`https://www.jsonrpc.org/`), JSONRPC "is transport agnostic in that the concepts can be used within the same process, over sockets, over http, or in many various message passing environments."

To model this agnosticism, the `jsonrpc` library uses objects of a `jsonrpc-connection` class, which represent a connection to a remote JSON endpoint (for details on Emacs's object system, see *EIEIO*). In modern object-oriented parlance, this class is "abstract", i.e. the actual class of a useful connection object is always a subclass of `jsonrpc-connection`. Nevertheless, we can define two distinct APIs around the `jsonrpc-connection` class:

1. A user interface for building JSONRPC applications

   In this scenario, the JSONRPC application selects a concrete subclass of `jsonrpc-connection`, and proceeds to create objects of that subclass using `make-instance`. To initiate a contact to the remote endpoint, the JSONRPC application passes this object to the functions `jsonrpc-notify`, `jsonrpc-request`, and/or `jsonrpc-async-request`. For handling remotely initiated contacts, which generally come in asynchronously, the instantiation should include `:request-dispatcher` and `:notification-dispatcher` initargs, which are both functions of 3 arguments: the connection object; a symbol naming the JSONRPC method invoked remotely; and a JSONRPC `params` object.

   The function passed as `:request-dispatcher` is responsible for handling the remote endpoint's requests, which expect a reply from the local endpoint (in this case, the program you're building). Inside that function, you may either return locally (a normal return) or non-locally (an error return). A local return value must be a Lisp object that can be serialized as JSON (see Section 33.31 [Parsing JSON], page 958). This determines a success response, and the object is forwarded to the server as the JSONRPC `result` object. A non-local return, achieved by calling the function `jsonrpc-error`, causes an error response to be sent to the server. The details of the accompanying JSONRPC `error` are filled out with whatever was passed to `jsonrpc-error`. A non-local return triggered by an unexpected error of any other type also causes an error response to be sent (unless you have set `debug-on-error`, in which case this calls the Lisp debugger, see Section 19.1.1 [Error Debugging], page 334).

2. A inheritance interface for building JSONRPC transport implementations

   In this scenario, `jsonrpc-connection` is subclassed to implement a different underlying transport strategy (for details on how to subclass, see Section "Inheritance" in `eieio`.). Users of the application-building interface can then instantiate objects of this concrete class (using the `make-instance` function) and connect to JSONRPC endpoints using that strategy.

   This API has mandatory and optional parts.

   To allow its users to initiate JSONRPC contacts (notifications or requests) or reply to endpoint requests, the subclass must have an implementation of the `jsonrpc-connection-send` method.

   Likewise, for handling the three types of remote contacts (requests, notifications, and responses to local requests), the transport implementation must arrange for the function `jsonrpc-connection-receive` to be called after noticing a new JSONRPC message on the wire (whatever that `"wire"` may be).

Finally, and optionally, the `jsonrpc-connection` subclass should implement the `jsonrpc-shutdown` and `jsonrpc-running-p` methods if these concepts apply to the transport. If they do, then any system resources (e.g. processes, timers, etc.) used to listen for messages on the wire should be released in `jsonrpc-shutdown`, i.e. they should only be needed while `jsonrpc-running-p` is non-`nil`.

33.32.2 Process-based JSONRPC connections

For convenience, the `jsonrpc` library comes with a built-in `jsonrpc-process-connection` transport implementation that can talk to local subprocesses (using the standard input and standard output); or TCP hosts (using sockets); or any other remote endpoint that Emacs's process object can represent (see Chapter 40 [Processes], page 1081).

Using this transport, the JSONRPC messages are encoded on the wire as plain text and prefaced by some basic HTTP-style enveloping headers, such as "Content-Length".

For an example of an application using this transport scheme on top of JSONRPC, see the Language Server Protocol (`https://microsoft.github.io/language-server-protocol/specification`).

Along with the mandatory `:request-dispatcher` and `:notification-dispatcher` initargs, users of the `jsonrpc-process-connection` class should pass the following initargs as keyword-value pairs to `make-instance`:

`:process` Value must be a live process object or a function of no arguments producing one such object. If passed a process object, the object is expected to contain a pre-established connection; otherwise, the function is called immediately after the object is made.

`:on-shutdown`
Value must be a function of a single argument, the `jsonrpc-process-connection` object. The function is called after the underlying process object has been deleted (either deliberately by `jsonrpc-shutdown`, or unexpectedly, because of some external cause).

33.32.3 JSONRPC JSON object format

JSONRPC JSON objects are exchanged as Lisp plists (see Section 5.9 [Property Lists], page 99): JSON-compatible plists are handed to the dispatcher functions and, likewise, JSON-compatible plists should be given to `jsonrpc-notify`, `jsonrpc-request`, and `jsonrpc-async-request`.

To facilitate handling plists, this library makes liberal use of `cl-lib` library (see *Common Lisp Extensions for GNU Emacs Lisp*) and suggests (but doesn't force) its clients to do the same. A macro `jsonrpc-lambda` can be used to create a lambda for destructuring a JSON-object like in this example:

```
(jsonrpc-async-request
 myproc :frobnicate `(:foo "trix")
 :success-fn (jsonrpc-lambda (&key bar baz &allow-other-keys)
               (message "Server replied back with %s and %s!"
                        bar baz))
 :error-fn (jsonrpc-lambda (&key code message _data)
             (message "Sadly, server reports %s: %s"
```

```
code message)))
```

33.32.4 Deferred JSONRPC requests

In many RPC situations, synchronization between the two communicating endpoints is a matter of correctly designing the RPC application: when synchronization is needed, requests (which are blocking) should be used; when it isn't, notifications should suffice. However, when Emacs acts as one of these endpoints, asynchronous events (e.g. timer- or process-related) may be triggered while there is still uncertainty about the state of the remote endpoint. Furthermore, acting on these events may only sometimes demand synchronization, depending on the event's specific nature.

The `:deferred` keyword argument to `jsonrpc-request` and `jsonrpc-async-request` is designed to let the caller indicate that the specific request needs synchronization and its actual issuance may be delayed to the future, until some condition is satisfied. Specifying `:deferred` for a request doesn't mean it *will* be delayed, only that it *can* be. If the request isn't sent immediately, `jsonrpc` will make renewed efforts to send it at certain key times during communication, such as when receiving or sending other messages to the endpoint.

Before any attempt to send the request, the application-specific conditions are checked. Since the `jsonrpc` library can't know what these conditions are, the program can use the `jsonrpc-connection-ready-p` generic function (see Section 13.8 [Generic Functions], page 245) to specify them. The default method for this function returns `t`, but you can add overriding methods that return `nil` in some situations, based on the arguments passed to it, which are the `jsonrpc-connection` object (see Section 33.32.1 [JSONRPC Overview], page 960) and whichever value you passed as the `:deferred` keyword argument.

33.33 Atomic Change Groups

In database terminology, an *atomic* change is an indivisible change—it can succeed entirely or it can fail entirely, but it cannot partly succeed. A Lisp program can make a series of changes to one or several buffers as an *atomic change group*, meaning that either the entire series of changes will be installed in their buffers or, in case of an error, none of them will be.

To do this for one buffer, the one already current, simply write a call to `atomic-change-group` around the code that makes the changes, like this:

```
(atomic-change-group
  (insert foo)
  (delete-region x y))
```

If an error (or other nonlocal exit) occurs inside the body of `atomic-change-group`, it unmakes all the changes in that buffer that were during the execution of the body. This kind of change group has no effect on any other buffers—any such changes remain.

If you need something more sophisticated, such as to make changes in various buffers constitute one atomic group, you must directly call lower-level functions that `atomic-change-group` uses.

`prepare-change-group` **&optional** *buffer* [Function]

This function sets up a change group for buffer *buffer*, which defaults to the current buffer. It returns a handle that represents the change group. You must use this handle to activate the change group and subsequently to finish it.

To use the change group, you must *activate* it. You must do this before making any changes in the text of *buffer*.

`activate-change-group` *handle* [Function]
This function activates the change group that *handle* designates.

After you activate the change group, any changes you make in that buffer become part of it. Once you have made all the desired changes in the buffer, you must *finish* the change group. There are two ways to do this: you can either accept (and finalize) all the changes, or cancel them all.

`accept-change-group` *handle* [Function]
This function accepts all the changes in the change group specified by *handle*, making them final.

`cancel-change-group` *handle* [Function]
This function cancels and undoes all the changes in the change group specified by *handle*.

You can cause some or all of the changes in a change group to be considered as a single unit for the purposes of the `undo` commands (see Section 33.9 [Undo], page 898) by using `undo-amalgamate-change-group`.

`undo-amalgamate-change-group` [Function]
Amalgamate all the changes made in the change-group since the state identified by *handle*. This function removes all undo boundaries between undo records of changes since the state described by *handle*. Usually, *handle* is the handle returned by `prepare-change-group`, in which case all the changes since the beginning of the change-group are amalgamated into a single undo unit.

Your code should use `unwind-protect` to make sure the group is always finished. The call to `activate-change-group` should be inside the `unwind-protect`, in case the user types `C-g` just after it runs. (This is one reason why `prepare-change-group` and `activate-change-group` are separate functions, because normally you would call `prepare-change-group` before the start of that `unwind-protect`.) Once you finish the group, don't use the handle again—in particular, don't try to finish the same group twice.

To make a multibuffer change group, call `prepare-change-group` once for each buffer you want to cover, then use `nconc` to combine the returned values, like this:

```
(nconc (prepare-change-group buffer-1)
       (prepare-change-group buffer-2))
```

You can then activate the multibuffer change group with a single call to `activate-change-group`, and finish it with a single call to `accept-change-group` or `cancel-change-group`.

Nested use of several change groups for the same buffer works as you would expect. Non-nested use of change groups for the same buffer will get Emacs confused, so don't let it happen; the first change group you start for any given buffer should be the last one finished.

33.34 Change Hooks

These hook variables let you arrange to take notice of changes in buffers (or in a particular buffer, if you make them buffer-local). See also Section 33.19.4 [Special Properties], page 927, for how to detect changes to specific parts of the text.

The functions you use in these hooks should save and restore the match data if they do anything that uses regular expressions; otherwise, they will interfere in bizarre ways with the editing operations that call them.

`before-change-functions` [Variable]

This variable holds a list of functions to call when Emacs is about to modify a buffer. Each function gets two arguments, the beginning and end of the region that is about to change, represented as integers. The buffer that is about to change is always the current buffer when the function is called.

`after-change-functions` [Variable]

This variable holds a list of functions to call after Emacs modifies a buffer. Each function receives three arguments: the beginning and end of the region just changed, and the length of the text that existed before the change. All three arguments are integers. The buffer that has been changed is always the current buffer when the function is called.

The length of the old text is the difference between the buffer positions before and after that text as it was before the change. As for the changed text, its length is simply the difference between the first two arguments.

Output of messages into the `*Messages*` buffer does not call these functions, and neither do certain internal buffer changes, such as changes in buffers created by Emacs internally for certain jobs, that should not be visible to Lisp programs.

The vast majority of buffer changing primitives will call `before-change-functions` and `after-change-functions` in balanced pairs, once for each change, where the arguments to these hooks exactly delimit the change being made. Yet, hook functions should not rely on this always being the case, because some complex primitives call `before-change-functions` once before making changes, and then call `after-change-functions` zero or more times, depending on how many individual changes the primitive is making. When that happens, the arguments to `before-change-functions` will enclose a region in which the individual changes are made, but won't necessarily be the minimal such region, and the arguments to each successive call of `after-change-functions` will then delimit the part of text being changed exactly. In general, we advise using either the before- or the after-change hook, but not both.

`combine-after-change-calls` *body...* [Macro]

The macro executes *body* normally, but arranges to call the after-change functions just once for a series of several changes—if that seems safe.

If a program makes several text changes in the same area of the buffer, using the macro `combine-after-change-calls` around that part of the program can make it run considerably faster when after-change hooks are in use. When the after-change hooks are ultimately called, the arguments specify a portion of the buffer including all of the changes made within the `combine-after-change-calls` body.

Warning: You must not alter the values of `after-change-functions` within the body of a `combine-after-change-calls` form.

Warning: If the changes you combine occur in widely scattered parts of the buffer, this will still work, but it is not advisable, because it may lead to inefficient behavior for some change hook functions.

`combine-change-calls` *beg end body...* [Macro]

This executes *body* normally, except any buffer changes it makes do not trigger the calls to `before-change-functions` and `after-change-functions`. Instead there is a single call of each of these hooks for the region enclosed by *beg* and *end*, the parameters supplied to `after-change-functions` reflecting the changes made to the size of the region by *body*.

The result of this macro is the result returned by *body*.

This macro is useful when a function makes a possibly large number of repetitive changes to the buffer, and the change hooks would otherwise take a long time to run, were they to be run for each individual buffer modification. Emacs itself uses this macro, for example, in the commands `comment-region` and `uncomment-region`.

Warning: You must not alter the values of `before-change-functions` or `after-change-function` within *body*.

Warning: You must not make any buffer changes outside of the region specified by *beg* and *end*.

`first-change-hook` [Variable]

This variable is a normal hook that is run whenever a buffer is changed that was previously in the unmodified state.

`inhibit-modification-hooks` [Variable]

If this variable is non-`nil`, all of the change hooks are disabled; none of them run. This affects all the hook variables described above in this section, as well as the hooks attached to certain special text properties (see Section 33.19.4 [Special Properties], page 927) and overlay properties (see Section 41.9.2 [Overlay Properties], page 1157).

Also, this variable is bound to non-`nil` while running those same hook variables, so that by default modifying the buffer from a modification hook does not cause other modification hooks to be run. If you do want modification hooks to be run in a particular piece of code that is itself run from a modification hook, then rebind locally `inhibit-modification-hooks` to `nil`. However, doing this may cause recursive calls to the modification hooks, so be sure to prepare for that (for example, by binding some variable which tells your hook to do nothing).

We recommend that you only bind this variable for modifications that do not result in lasting changes to buffer text contents (for example face changes or temporary modifications). If you need to delay change hooks during a series of changes (typically for performance reasons), use `combine-change-calls` or `combine-after-change-calls` instead.

34 Non-ASCII Characters

This chapter covers the special issues relating to characters and how they are stored in strings and buffers.

34.1 Text Representations

Emacs buffers and strings support a large repertoire of characters from many different scripts, allowing users to type and display text in almost any known written language.

To support this multitude of characters and scripts, Emacs closely follows the *Unicode Standard*. The Unicode Standard assigns a unique number, called a *codepoint*, to each and every character. The range of codepoints defined by Unicode, or the Unicode *codespace*, is `0..#x10FFFF` (in hexadecimal notation), inclusive. Emacs extends this range with codepoints in the range `#x110000..#x3FFFFF`, which it uses for representing characters that are not unified with Unicode and *raw 8-bit bytes* that cannot be interpreted as characters. Thus, a character codepoint in Emacs is a 22-bit integer.

To conserve memory, Emacs does not hold fixed-length 22-bit numbers that are codepoints of text characters within buffers and strings. Rather, Emacs uses a variable-length internal representation of characters, that stores each character as a sequence of 1 to 5 8-bit bytes, depending on the magnitude of its codepoint[1]. For example, any ASCII character takes up only 1 byte, a Latin-1 character takes up 2 bytes, etc. We call this representation of text *multibyte*.

Outside Emacs, characters can be represented in many different encodings, such as ISO-8859-1, GB-2312, Big-5, etc. Emacs converts between these external encodings and its internal representation, as appropriate, when it reads text into a buffer or a string, or when it writes text to a disk file or passes it to some other process.

Occasionally, Emacs needs to hold and manipulate encoded text or binary non-text data in its buffers or strings. For example, when Emacs visits a file, it first reads the file's text verbatim into a buffer, and only then converts it to the internal representation. Before the conversion, the buffer holds encoded text.

Encoded text is not really text, as far as Emacs is concerned, but rather a sequence of raw 8-bit bytes. We call buffers and strings that hold encoded text *unibyte* buffers and strings, because Emacs treats them as a sequence of individual bytes. Usually, Emacs displays unibyte buffers and strings as octal codes such as `\237`. We recommend that you never use unibyte buffers and strings except for manipulating encoded text or binary non-text data.

In a buffer, the buffer-local value of the variable `enable-multibyte-characters` specifies the representation used. The representation for a string is determined and recorded in the string when the string is constructed.

`enable-multibyte-characters` [Variable]

This variable specifies the current buffer's text representation. If it is non-`nil`, the buffer contains multibyte text; otherwise, it contains unibyte encoded text or binary non-text data.

[1] This internal representation is based on one of the encodings defined by the Unicode Standard, called *UTF-8*, for representing any Unicode codepoint, but Emacs extends UTF-8 to represent the additional codepoints it uses for raw 8-bit bytes and characters not unified with Unicode.

You cannot set this variable directly; instead, use the function `set-buffer-multibyte` to change a buffer's representation.

`position-bytes` *position* [Function]
Buffer positions are measured in character units. This function returns the byte-position corresponding to buffer position *position* in the current buffer. This is 1 at the start of the buffer, and counts upward in bytes. If *position* is out of range, the value is `nil`.

`byte-to-position` *byte-position* [Function]
Return the buffer position, in character units, corresponding to given *byte-position* in the current buffer. If *byte-position* is out of range, the value is `nil`. In a multibyte buffer, an arbitrary value of *byte-position* can be not at character boundary, but inside a multibyte sequence representing a single character; in this case, this function returns the buffer position of the character whose multibyte sequence includes *byte-position*. In other words, the value does not change for all byte positions that belong to the same character.

The following two functions are useful when a Lisp program needs to map buffer positions to byte offsets in a file visited by the buffer.

`bufferpos-to-filepos` *position* **&optional** *quality coding-system* [Function]
This function is similar to `position-bytes`, but instead of byte position in the current buffer it returns the offset from the beginning of the current buffer's file of the byte that corresponds to the given character *position* in the buffer. The conversion requires to know how the text is encoded in the buffer's file; this is what the *coding-system* argument is for, defaulting to the value of `buffer-file-coding-system`. The optional argument *quality* specifies how accurate the result should be; it should be one of the following:

`exact` The result must be accurate. The function may need to encode and decode a large part of the buffer, which is expensive and can be slow.

`approximate`
The value can be an approximation. The function may avoid expensive processing and return an inexact result.

`nil` If the exact result needs expensive processing, the function will return `nil` rather than an approximation. This is the default if the argument is omitted.

`filepos-to-bufferpos` *byte* **&optional** *quality coding-system* [Function]
This function returns the buffer position corresponding to a file position specified by *byte*, a zero-base byte offset from the file's beginning. The function performs the conversion opposite to what `bufferpos-to-filepos` does. Optional arguments *quality* and *coding-system* have the same meaning and values as for `bufferpos-to-filepos`.

`multibyte-string-p` *string* [Function]
Return `t` if *string* is a multibyte string, `nil` otherwise. This function also returns `nil` if *string* is some object other than a string.

`string-bytes` *string* [Function]
: This function returns the number of bytes in *string*. If *string* is a multibyte string, this can be greater than `(length` *string*`)`.

`unibyte-string` **&rest** *bytes* [Function]
: This function concatenates all its argument *bytes* and makes the result a unibyte string.

34.2 Disabling Multibyte Characters

By default, Emacs starts in multibyte mode: it stores the contents of buffers and strings using an internal encoding that represents non-ASCII characters using multi-byte sequences. Multibyte mode allows you to use all the supported languages and scripts without limitations.

Under very special circumstances, you may want to disable multibyte character support, for a specific buffer. When multibyte characters are disabled in a buffer, we call that *unibyte mode*. In unibyte mode, each character in the buffer has a character code ranging from 0 through 255 (0377 octal); 0 through 127 (0177 octal) represent ASCII characters, and 128 (0200 octal) through 255 (0377 octal) represent non-ASCII characters.

To edit a particular file in unibyte representation, visit it using `find-file-literally`. See Section 26.1.1 [Visiting Functions], page 610. You can convert a multibyte buffer to unibyte by saving it to a file, killing the buffer, and visiting the file again with `find-file-literally`. Alternatively, you can use *C-x* `RET` *c* (`universal-coding-system-argument`) and specify '`raw-text`' as the coding system with which to visit or save a file. See Section "Specifying a Coding System for File Text" in *GNU Emacs Manual*. Unlike `find-file-literally`, finding a file as '`raw-text`' doesn't disable format conversion, uncompression, or auto mode selection.

The buffer-local variable `enable-multibyte-characters` is non-`nil` in multibyte buffers, and `nil` in unibyte ones. The mode line also indicates whether a buffer is multibyte or not. With a graphical display, in a multibyte buffer, the portion of the mode line that indicates the character set has a tooltip that (amongst other things) says that the buffer is multibyte. In a unibyte buffer, the character set indicator is absent. Thus, in a unibyte buffer (when using a graphical display) there is normally nothing before the indication of the visited file's end-of-line convention (colon, backslash, etc.), unless you are using an input method.

You can turn off multibyte support in a specific buffer by invoking the command `toggle-enable-multibyte-characters` in that buffer.

34.3 Converting Text Representations

Emacs can convert unibyte text to multibyte; it can also convert multibyte text to unibyte, provided that the multibyte text contains only ASCII and 8-bit raw bytes. In general, these conversions happen when inserting text into a buffer, or when putting text from several strings together in one string. You can also explicitly convert a string's contents to either representation.

Emacs chooses the representation for a string based on the text from which it is constructed. The general rule is to convert unibyte text to multibyte text when combining it

with other multibyte text, because the multibyte representation is more general and can hold whatever characters the unibyte text has.

When inserting text into a buffer, Emacs converts the text to the buffer's representation, as specified by `enable-multibyte-characters` in that buffer. In particular, when you insert multibyte text into a unibyte buffer, Emacs converts the text to unibyte, even though this conversion cannot in general preserve all the characters that might be in the multibyte text. The other natural alternative, to convert the buffer contents to multibyte, is not acceptable because the buffer's representation is a choice made by the user that cannot be overridden automatically.

Converting unibyte text to multibyte text leaves ASCII characters unchanged, and converts bytes with codes 128 through 255 to the multibyte representation of raw eight-bit bytes.

Converting multibyte text to unibyte converts all ASCII and eight-bit characters to their single-byte form, but loses information for non-ASCII characters by discarding all but the low 8 bits of each character's codepoint. Converting unibyte text to multibyte and back to unibyte reproduces the original unibyte text.

The next two functions either return the argument *string*, or a newly created string with no text properties.

`string-to-multibyte` *string* [Function]
: This function returns a multibyte string containing the same sequence of characters as *string*. If *string* is a multibyte string, it is returned unchanged. The function assumes that *string* includes only ASCII characters and raw 8-bit bytes; the latter are converted to their multibyte representation corresponding to the codepoints `#x3FFF80` through `#x3FFFFF`, inclusive (see Section 34.1 [Text Representations], page 966).

`string-to-unibyte` *string* [Function]
: This function returns a unibyte string containing the same sequence of characters as *string*. If *string* is a unibyte string, it is returned unchanged. Otherwise, ASCII characters and characters in the `eight-bit` charset are converted to their corresponding byte values. Use this function for *string* arguments that contain only ASCII and eight-bit characters; the function signals an error if any other characters are encountered.

`byte-to-string` *byte* [Function]
: This function returns a unibyte string containing a single byte of character data, *byte*. It signals an error if *byte* is not an integer between 0 and 255.

`multibyte-char-to-unibyte` *char* [Function]
: This converts the multibyte character *char* to a unibyte character, and returns that character. If *char* is neither ASCII nor eight-bit, the function returns −1.

`unibyte-char-to-multibyte` *char* [Function]
: This converts the unibyte character *char* to a multibyte character, assuming *char* is either ASCII or raw 8-bit byte.

34.4 Selecting a Representation

Sometimes it is useful to examine an existing buffer or string as multibyte when it was unibyte, or vice versa.

`set-buffer-multibyte` *multibyte* [Function]

Set the representation type of the current buffer. If *multibyte* is non-`nil`, the buffer becomes multibyte. If *multibyte* is `nil`, the buffer becomes unibyte.

This function leaves the buffer contents unchanged when viewed as a sequence of bytes. As a consequence, it can change the contents viewed as characters; for instance, a sequence of three bytes which is treated as one character in multibyte representation will count as three characters in unibyte representation. Eight-bit characters representing raw bytes are an exception. They are represented by one byte in a unibyte buffer, but when the buffer is set to multibyte, they are converted to two-byte sequences, and vice versa.

This function sets `enable-multibyte-characters` to record which representation is in use. It also adjusts various data in the buffer (including overlays, text properties and markers) so that they cover the same text as they did before.

This function signals an error if the buffer is narrowed, since the narrowing might have occurred in the middle of multibyte character sequences.

This function also signals an error if the buffer is an indirect buffer. An indirect buffer always inherits the representation of its base buffer.

`string-as-unibyte` *string* [Function]

If *string* is already a unibyte string, this function returns *string* itself. Otherwise, it returns a new string with the same bytes as *string*, but treating each byte as a separate character (so that the value may have more characters than *string*); as an exception, each eight-bit character representing a raw byte is converted into a single byte. The newly-created string contains no text properties.

`string-as-multibyte` *string* [Function]

If *string* is a multibyte string, this function returns *string* itself. Otherwise, it returns a new string with the same bytes as *string*, but treating each multibyte sequence as one character. This means that the value may have fewer characters than *string* has. If a byte sequence in *string* is invalid as a multibyte representation of a single character, each byte in the sequence is treated as a raw 8-bit byte. The newly-created string contains no text properties.

34.5 Character Codes

The unibyte and multibyte text representations use different character codes. The valid character codes for unibyte representation range from 0 to `#xFF` (255)—the values that can fit in one byte. The valid character codes for multibyte representation range from 0 to `#x3FFFFF`. In this code space, values 0 through `#x7F` (127) are for ASCII characters, and values `#x80` (128) through `#x3FFF7F` (4194175) are for non-ASCII characters.

Emacs character codes are a superset of the Unicode standard. Values 0 through `#x10FFFF` (1114111) correspond to Unicode characters of the same codepoint; values `#x110000` (1114112) through `#x3FFF7F` (4194175) represent characters that are not unified

with Unicode; and values `#x3FFF80` (4194176) through `#x3FFFFF` (4194303) represent eight-bit raw bytes.

`characterp` *charcode* [Function]

This returns `t` if *charcode* is a valid character, and `nil` otherwise.

```
(characterp 65)
     ⇒ t
(characterp 4194303)
     ⇒ t
(characterp 4194304)
     ⇒ nil
```

`max-char` **&optional** *unicode* [Function]

This function returns the largest value that a valid character codepoint can have in Emacs. If the optional argument *unicode* is non-`nil`, it returns the largest character codepoint defined by the Unicode Standard (which is smaller than the maximum codepoint supported by Emacs).

```
(characterp (max-char))
     ⇒ t
(characterp (1+ (max-char)))
     ⇒ nil
```

`char-from-name` *string* **&optional** *ignore-case* [Function]

This function returns the character whose Unicode name is *string*. If *ignore-case* is non-`nil`, case is ignored in *string*. This function returns `nil` if *string* does not name a character.

```
;; U+03A3
(= (char-from-name "GREEK CAPITAL LETTER SIGMA") #x03A3)
     ⇒ t
```

`get-byte` **&optional** *pos string* [Function]

This function returns the byte at character position *pos* in the current buffer. If the current buffer is unibyte, this is literally the byte at that position. If the buffer is multibyte, byte values of ASCII characters are the same as character codepoints, whereas eight-bit raw bytes are converted to their 8-bit codes. The function signals an error if the character at *pos* is non-ASCII.

The optional argument *string* means to get a byte value from that string instead of the current buffer.

34.6 Character Properties

A *character property* is a named attribute of a character that specifies how the character behaves and how it should be handled during text processing and display. Thus, character properties are an important part of specifying the character's semantics.

On the whole, Emacs follows the Unicode Standard in its implementation of character properties. In particular, Emacs supports the Unicode Character Property Model (`https://www.unicode.org/reports/tr23/`), and the Emacs character property database

is derived from the Unicode Character Database (UCD). See the Character Properties chapter of the Unicode Standard (`https://www.unicode.org/versions/Unicode15.0.0/ch04.pdf`), for a detailed description of Unicode character properties and their meaning. This section assumes you are already familiar with that chapter of the Unicode Standard, and want to apply that knowledge to Emacs Lisp programs.

In Emacs, each property has a name, which is a symbol, and a set of possible values, whose types depend on the property; if a character does not have a certain property, the value is `nil`. As a general rule, the names of character properties in Emacs are produced from the corresponding Unicode properties by downcasing them and replacing each '`_`' character with a dash '`-`'. For example, `Canonical_Combining_Class` becomes `canonical-combining-class`. However, sometimes we shorten the names to make their use easier.

Some codepoints are left *unassigned* by the UCD—they don't correspond to any character. The Unicode Standard defines default values of properties for such codepoints; they are mentioned below for each property.

Here is the full list of value types for all the character properties that Emacs knows about:

`name`
: Corresponds to the `Name` Unicode property. The value is a string consisting of upper-case Latin letters A to Z, digits, spaces, and hyphen '`-`' characters. For unassigned codepoints, the value is `nil`.

`general-category`
: Corresponds to the `General_Category` Unicode property. The value is a symbol whose name is a 2-letter abbreviation of the character's classification. For unassigned codepoints, the value is `Cn`.

`canonical-combining-class`
: Corresponds to the `Canonical_Combining_Class` Unicode property. The value is an integer. For unassigned codepoints, the value is zero.

`bidi-class`
: Corresponds to the Unicode `Bidi_Class` property. The value is a symbol whose name is the Unicode *directional type* of the character. Emacs uses this property when it reorders bidirectional text for display (see Section 41.27 [Bidirectional Display], page 1255). For unassigned codepoints, the value depends on the code blocks to which the codepoint belongs: most unassigned codepoints get the value of `L` (strong L), but some get values of `AL` (Arabic letter) or `R` (strong R).

`decomposition`
: Corresponds to the Unicode properties `Decomposition_Type` and `Decomposition_Value`. The value is a list, whose first element may be a symbol representing a compatibility formatting tag, such as `small`[2]; the other elements are characters that give the compatibility decomposition sequence of this character. For characters that don't have decomposition sequences,

[2] The Unicode specification writes these tag names inside '`<..>`' brackets, but the tag names in Emacs do not include the brackets; e.g., Unicode specifies '`<small>`' where Emacs uses '`small`'.

and for unassigned codepoints, the value is a list with a single member, the character itself.

`decimal-digit-value`
: Corresponds to the Unicode `Numeric_Value` property for characters whose `Numeric_Type` is '`Decimal`'. The value is an integer, or `nil` if the character has no decimal digit value. For unassigned codepoints, the value is `nil`, which means NaN, or "not a number".

`digit-value`
: Corresponds to the Unicode `Numeric_Value` property for characters whose `Numeric_Type` is '`Digit`'. The value is an integer. Examples of such characters include compatibility subscript and superscript digits, for which the value is the corresponding number. For characters that don't have any numeric value, and for unassigned codepoints, the value is `nil`, which means NaN.

`numeric-value`
: Corresponds to the Unicode `Numeric_Value` property for characters whose `Numeric_Type` is '`Numeric`'. The value of this property is a number. Examples of characters that have this property include fractions, subscripts, superscripts, Roman numerals, currency numerators, and encircled numbers. For example, the value of this property for the character U+2155 VULGAR FRACTION ONE FIFTH is `0.2`. For characters that don't have any numeric value, and for unassigned codepoints, the value is `nil`, which means NaN.

`mirrored`
: Corresponds to the Unicode `Bidi_Mirrored` property. The value of this property is a symbol, either `Y` or `N`. For unassigned codepoints, the value is `N`.

`mirroring`
: Corresponds to the Unicode `Bidi_Mirroring_Glyph` property. The value of this property is a character whose glyph represents the mirror image of the character's glyph, or `nil` if there's no defined mirroring glyph. All the characters whose `mirrored` property is `N` have `nil` as their `mirroring` property; however, some characters whose `mirrored` property is `Y` also have `nil` for `mirroring`, because no appropriate characters exist with mirrored glyphs. Emacs uses this property to display mirror images of characters when appropriate (see Section 41.27 [Bidirectional Display], page 1255). For unassigned codepoints, the value is `nil`.

`paired-bracket`
: Corresponds to the Unicode `Bidi_Paired_Bracket` property. The value of this property is the codepoint of a character's *paired bracket*, or `nil` if the character is not a bracket character. This establishes a mapping between characters that are treated as bracket pairs by the Unicode Bidirectional Algorithm; Emacs uses this property when it decides how to reorder for display parentheses, braces, and other similar characters (see Section 41.27 [Bidirectional Display], page 1255).

`bracket-type`
: Corresponds to the Unicode `Bidi_Paired_Bracket_Type` property. For characters whose `paired-bracket` property is non-`nil`, the value of this property is a

symbol, either `o` (for opening bracket characters) or `c` (for closing bracket characters). For characters whose `paired-bracket` property is `nil`, the value is the symbol `n` (None). Like `paired-bracket`, this property is used for bidirectional display.

`old-name` Corresponds to the Unicode `Unicode_1_Name` property. The value is a string. For unassigned codepoints, and characters that have no value for this property, the value is `nil`.

`iso-10646-comment`
Corresponds to the Unicode `ISO_Comment` property. The value is either a string or `nil`. For unassigned codepoints, the value is `nil`.

`uppercase`
Corresponds to the Unicode `Simple_Uppercase_Mapping` property. The value of this property is a single character. For unassigned codepoints, the value is `nil`, which means the character itself.

`lowercase`
Corresponds to the Unicode `Simple_Lowercase_Mapping` property. The value of this property is a single character. For unassigned codepoints, the value is `nil`, which means the character itself.

`titlecase`
Corresponds to the Unicode `Simple_Titlecase_Mapping` property. *Title case* is a special form of a character used when the first character of a word needs to be capitalized. The value of this property is a single character. For unassigned codepoints, the value is `nil`, which means the character itself.

`special-uppercase`
Corresponds to Unicode language- and context-independent special upper-casing rules. The value of this property is a string (which may be empty). For example mapping for U+00DF LATIN SMALL LETTER SHARP S is `"SS"`. For characters with no special mapping, the value is `nil` which means `uppercase` property needs to be consulted instead.

`special-lowercase`
Corresponds to Unicode language- and context-independent special lower-casing rules. The value of this property is a string (which may be empty). For example mapping for U+0130 LATIN CAPITAL LETTER I WITH DOT ABOVE the value is `"i\u0307"` (i.e. 2-character string consisting of LATIN SMALL LETTER I followed by U+0307 COMBINING DOT ABOVE). For characters with no special mapping, the value is `nil` which means `lowercase` property needs to be consulted instead.

`special-titlecase`
Corresponds to Unicode unconditional special title-casing rules. The value of this property is a string (which may be empty). For example mapping for U+FB01 LATIN SMALL LIGATURE FI the value is `"Fi"`. For characters with no special mapping, the value is `nil` which means `titlecase` property needs to be consulted instead.

`get-char-code-property` *char propname* [Function]

This function returns the value of *char*'s *propname* property.

```
(get-char-code-property ?\s 'general-category)
     ⇒ Zs
(get-char-code-property ?1 'general-category)
     ⇒ Nd
;; U+2084
(get-char-code-property ?\N{SUBSCRIPT FOUR}
                        'digit-value)
     ⇒ 4
;; U+2155
(get-char-code-property ?\N{VULGAR FRACTION ONE FIFTH}
                        'numeric-value)
     ⇒ 0.2
;; U+2163
(get-char-code-property ?\N{ROMAN NUMERAL FOUR}
                        'numeric-value)
     ⇒ 4
(get-char-code-property ?\( 'paired-bracket)
     ⇒ 41  ; closing parenthesis
(get-char-code-property ?\) 'bracket-type)
     ⇒ c
```

`char-code-property-description` *prop value* [Function]

This function returns the description string of property *prop*'s *value*, or `nil` if *value* has no description.

```
(char-code-property-description 'general-category 'Zs)
     ⇒ "Separator, Space"
(char-code-property-description 'general-category 'Nd)
     ⇒ "Number, Decimal Digit"
(char-code-property-description 'numeric-value '1/5)
     ⇒ nil
```

`put-char-code-property` *char propname value* [Function]

This function stores *value* as the value of the property *propname* for the character *char*.

`unicode-category-table` [Variable]

The value of this variable is a char-table (see Section 6.6 [Char-Tables], page 119) that specifies, for each character, its Unicode `General_Category` property as a symbol.

`char-script-table` [Variable]

The value of this variable is a char-table that specifies, for each character, a symbol whose name is the script to which the character belongs, according to the Unicode Standard classification of the Unicode code space into script-specific blocks. This char-table has a single extra slot whose value is the list of all script symbols. Note that Emacs' classification of characters into scripts is not a 1-for-1 reflection of the Unicode standard, e.g. there is no '`symbol`' script in Unicode.

`char-width-table` [Variable]
: The value of this variable is a char-table that specifies the width of each character in columns that it will occupy on the screen.

`printable-chars` [Variable]
: The value of this variable is a char-table that specifies, for each character, whether it is printable or not. That is, if evaluating `(aref printable-chars char)` results in `t`, the character is printable, and if it results in `nil`, it is not.

34.7 Character Sets

An Emacs *character set*, or *charset*, is a set of characters in which each character is assigned a numeric code point. (The Unicode Standard calls this a *coded character set*.) Each Emacs charset has a name which is a symbol. A single character can belong to any number of different character sets, but it will generally have a different code point in each charset. Examples of character sets include `ascii`, `iso-8859-1`, `greek-iso8859-7`, and `windows-1255`. The code point assigned to a character in a charset is usually different from its code point used in Emacs buffers and strings.

Emacs defines several special character sets. The character set `unicode` includes all the characters whose Emacs code points are in the range `0..#x10FFFF`. The character set `emacs` includes all ASCII and non-ASCII characters. Finally, the `eight-bit` charset includes the 8-bit raw bytes; Emacs uses it to represent raw bytes encountered in text.

`charsetp` *object* [Function]
: Returns `t` if *object* is a symbol that names a character set, `nil` otherwise.

`charset-list` [Variable]
: The value is a list of all defined character set names.

`charset-priority-list` **&optional** *highestp* [Function]
: This function returns a list of all defined character sets ordered by their priority. If *highestp* is non-`nil`, the function returns a single character set of the highest priority.

`set-charset-priority` **&rest** *charsets* [Function]
: This function makes *charsets* the highest priority character sets.

`char-charset` *character* **&optional** *restriction* [Function]
: This function returns the name of the character set of highest priority that *character* belongs to. ASCII characters are an exception: for them, this function always returns `ascii`.

  If *restriction* is non-`nil`, it should be a list of charsets to search. Alternatively, it can be a coding system, in which case the returned charset must be supported by that coding system (see Section 34.10 [Coding Systems], page 980).

`charset-plist` *charset* [Function]
: This function returns the property list of the character set *charset*. Although *charset* is a symbol, this is not the same as the property list of that symbol. Charset properties include important information about the charset, such as its documentation string, short name, etc.

`put-charset-property` *charset propname value* [Function]
: This function sets the *propname* property of *charset* to the given *value*.

`get-charset-property` *charset propname* [Function]
: This function returns the value of *charset*s property *propname*.

`list-charset-chars` *charset* [Command]
: This command displays a list of characters in the character set *charset*.

Emacs can convert between its internal representation of a character and the character's codepoint in a specific charset. The following two functions support these conversions.

`decode-char` *charset code-point* [Function]
: This function decodes a character that is assigned a *code-point* in *charset*, to the corresponding Emacs character, and returns it. If *charset* doesn't contain a character of that code point, the value is `nil`.

  For backward compatibility, if *code-point* doesn't fit in a Lisp fixnum (see Section 3.1 [Integer Basics], page 38), it can be specified as a cons cell `(high . low)`, where *low* are the lower 16 bits of the value and *high* are the high 16 bits. This usage is obsolescent.

`encode-char` *char charset* [Function]
: This function returns the code point assigned to the character *char* in *charset*. If *charset* doesn't have a codepoint for *char*, the value is `nil`.

The following function comes in handy for applying a certain function to all or part of the characters in a charset:

`map-charset-chars` *function charset* **&optional** *arg from-code to-code* [Function]
: Call *function* for characters in *charset*. *function* is called with two arguments. The first one is a cons cell `(from . to)`, where *from* and *to* indicate a range of characters contained in *charset*. The second argument passed to *function* is *arg*, or `nil` if *arg* is omitted.

  By default, the range of codepoints passed to *function* includes all the characters in *charset*, but optional arguments *from-code* and *to-code* limit that to the range of characters between these two codepoints of *charset*. If either of them is `nil`, it defaults to the first or last codepoint of *charset*, respectively. Note that *from-code* and *to-code* are *charset*'s codepoints, not the Emacs codes of characters; by contrast, the values *from* and *to* in the cons cell passed to *function are* Emacs character codes. Those Emacs character codes are either Unicode code points, or Emacs internal code points that extend Unicode and are beyond the Unicode range of characters `0..#x10FFFF` (see Section 34.1 [Text Representations], page 966). The latter happens rarely, with legacy CJK charsets for codepoints of *charset* which specify characters not yet unified with Unicode.

34.8 Scanning for Character Sets

Sometimes it is useful to find out which character set a particular character belongs to. One use for this is in determining which coding systems (see Section 34.10 [Coding Systems], page 980) are capable of representing all of the text in question; another is to determine the font(s) for displaying that text.

`charset-after` **&optional** *pos* [Function]
: This function returns the charset of highest priority containing the character at position *pos* in the current buffer. If *pos* is omitted or `nil`, it defaults to the current value of point. If *pos* is out of range, the value is `nil`.

`find-charset-region` *beg end* **&optional** *translation* [Function]
: This function returns a list of the character sets of highest priority that contain characters in the current buffer between positions *beg* and *end*.

  The optional argument *translation* specifies a translation table to use for scanning the text (see Section 34.9 [Translation of Characters], page 978). If it is non-`nil`, then each character in the region is translated through this table, and the value returned describes the translated characters instead of the characters actually in the buffer.

`find-charset-string` *string* **&optional** *translation* [Function]
: This function returns a list of character sets of highest priority that contain characters in *string*. It is just like `find-charset-region`, except that it applies to the contents of *string* instead of part of the current buffer.

34.9 Translation of Characters

A *translation table* is a char-table (see Section 6.6 [Char-Tables], page 119) that specifies a mapping of characters into characters. These tables are used in encoding and decoding, and for other purposes. Some coding systems specify their own particular translation tables; there are also default translation tables which apply to all other coding systems.

A translation table has two extra slots. The first is either `nil` or a translation table that performs the reverse translation; the second is the maximum number of characters to look up for translating sequences of characters (see the description of `make-translation-table-from-alist` below).

`make-translation-table` **&rest** *translations* [Function]
: This function returns a translation table based on the argument *translations*. Each element of *translations* should be a list of elements of the form `(`*`from . to`*`)`; this says to translate the character *from* into *to*.

  The arguments and the forms in each argument are processed in order, and if a previous form already translates *to* to some other character, say *to-alt*, *from* is also translated to *to-alt*.

During decoding, the translation table's translations are applied to the characters that result from ordinary decoding. If a coding system has the property `:decode-translation-table`, that specifies the translation table to use, or a list of translation tables to apply in sequence. (This is a property of the coding system, as returned by `coding-system-get`, not a property of the symbol that is the coding system's name. See Section 34.10.1 [Basic

Concepts of Coding Systems], page 980.) Finally, if `standard-translation-table-for-decode` is non-`nil`, the resulting characters are translated by that table.

During encoding, the translation table's translations are applied to the characters in the buffer, and the result of translation is actually encoded. If a coding system has property `:encode-translation-table`, that specifies the translation table to use, or a list of translation tables to apply in sequence. In addition, if the variable `standard-translation-table-for-encode` is non-`nil`, it specifies the translation table to use for translating the result.

`standard-translation-table-for-decode` [Variable]
: This is the default translation table for decoding. If a coding system specifies its own translation tables, the table that is the value of this variable, if non-`nil`, is applied after them.

`standard-translation-table-for-encode` [Variable]
: This is the default translation table for encoding. If a coding system specifies its own translation tables, the table that is the value of this variable, if non-`nil`, is applied after them.

`translation-table-for-input` [Variable]
: Self-inserting characters are translated through this translation table before they are inserted. Search commands also translate their input through this table, so they can compare more reliably with what's in the buffer.

  This variable automatically becomes buffer-local when set.

`make-translation-table-from-vector` *vec* [Function]
: This function returns a translation table made from *vec* that is an array of 256 elements to map bytes (values 0 through #xFF) to characters. Elements may be `nil` for untranslated bytes. The returned table has a translation table for reverse mapping in the first extra slot, and the value `1` in the second extra slot.

  This function provides an easy way to make a private coding system that maps each byte to a specific character. You can specify the returned table and the reverse translation table using the properties `:decode-translation-table` and `:encode-translation-table` respectively in the *props* argument to `define-coding-system`.

`make-translation-table-from-alist` *alist* [Function]
: This function is similar to `make-translation-table` but returns a complex translation table rather than a simple one-to-one mapping. Each element of *alist* is of the form `(`*from* `.` *to*`)`, where *from* and *to* are either characters or vectors specifying a sequence of characters. If *from* is a character, that character is translated to *to* (i.e., to a character or a character sequence). If *from* is a vector of characters, that sequence is translated to *to*. The returned table has a translation table for reverse mapping in the first extra slot, and the maximum length of all the *from* character sequences in the second extra slot.

34.10 Coding Systems

When Emacs reads or writes a file, and when Emacs sends text to a subprocess or receives text from a subprocess, it normally performs character code conversion and end-of-line conversion as specified by a particular *coding system*.

How to define a coding system is an arcane matter, and is not documented here.

34.10.1 Basic Concepts of Coding Systems

Character code conversion involves conversion between the internal representation of characters used inside Emacs and some other encoding. Emacs supports many different encodings, in that it can convert to and from them. For example, it can convert text to or from encodings such as Latin 1, Latin 2, Latin 3, Latin 4, Latin 5, and several variants of ISO 2022. In some cases, Emacs supports several alternative encodings for the same characters; for example, there are three coding systems for the Cyrillic (Russian) alphabet: ISO, Alternativnyj, and KOI8.

Every coding system specifies a particular set of character code conversions, but the coding system `undecided` is special: it leaves the choice unspecified, to be chosen heuristically for each file or string, based on the file's or string's data, when they are decoded or encoded. The coding system `prefer-utf-8` is like `undecided`, but it prefers to choose `utf-8` when possible.

In general, a coding system doesn't guarantee roundtrip identity: decoding a byte sequence using a coding system, then encoding the resulting text in the same coding system, can produce a different byte sequence. But some coding systems do guarantee that the byte sequence will be the same as what you originally decoded. Here are a few examples:

iso-8859-1, utf-8, big5, shift_jis, euc-jp

Encoding buffer text and then decoding the result can also fail to reproduce the original text. For instance, if you encode a character with a coding system which does not support that character, the result is unpredictable, and thus decoding it using the same coding system may produce a different text. Currently, Emacs can't report errors that result from encoding unsupported characters.

End of line conversion handles three different conventions used on various systems for representing end of line in files. The Unix convention, used on GNU and Unix systems, is to use the linefeed character (also called newline). The DOS convention, used on MS-Windows and MS-DOS systems, is to use a carriage return and a linefeed at the end of a line. The Mac convention is to use just carriage return. (This was the convention used in Classic Mac OS.)

Base coding systems such as `latin-1` leave the end-of-line conversion unspecified, to be chosen based on the data. *Variant coding systems* such as `latin-1-unix`, `latin-1-dos` and `latin-1-mac` specify the end-of-line conversion explicitly as well. Most base coding systems have three corresponding variants whose names are formed by adding '`-unix`', '`-dos`' and '`-mac`'.

The coding system `raw-text` is special in that it prevents character code conversion, and causes the buffer visited with this coding system to be a unibyte buffer. For historical reasons, you can save both unibyte and multibyte text with this coding system. When you use `raw-text` to encode multibyte text, it does perform one character code conversion: it

converts eight-bit characters to their single-byte external representation. `raw-text` does not specify the end-of-line conversion, allowing that to be determined as usual by the data, and has the usual three variants which specify the end-of-line conversion.

`no-conversion` (and its alias `binary`) is equivalent to `raw-text-unix`: it specifies no conversion of either character codes or end-of-line.

The coding system `utf-8-emacs` specifies that the data is represented in the internal Emacs encoding (see Section 34.1 [Text Representations], page 966). This is like `raw-text` in that no code conversion happens, but different in that the result is multibyte data. The name `emacs-internal` is an alias for `utf-8-emacs-unix` (so it forces no conversion of end-of-line, unlike `utf-8-emacs`, which can decode all 3 kinds of end-of-line conventions).

`coding-system-get` *coding-system property* [Function]

This function returns the specified property of the coding system *coding-system*. Most coding system properties exist for internal purposes, but one that you might find useful is `:mime-charset`. That property's value is the name used in MIME for the character coding which this coding system can read and write. Examples:

```
(coding-system-get 'iso-latin-1 :mime-charset)
     ⇒ iso-8859-1
(coding-system-get 'iso-2022-cn :mime-charset)
     ⇒ iso-2022-cn
(coding-system-get 'cyrillic-koi8 :mime-charset)
     ⇒ koi8-r
```

The value of the `:mime-charset` property is also defined as an alias for the coding system.

`coding-system-aliases` *coding-system* [Function]

This function returns the list of aliases of *coding-system*.

34.10.2 Encoding and I/O

The principal purpose of coding systems is for use in reading and writing files. The function `insert-file-contents` uses a coding system to decode the file data, and `write-region` uses one to encode the buffer contents.

You can specify the coding system to use either explicitly (see Section 34.10.6 [Specifying Coding Systems], page 989), or implicitly using a default mechanism (see Section 34.10.5 [Default Coding Systems], page 986). But these methods may not completely specify what to do. For example, they may choose a coding system such as `undecided` which leaves the character code conversion to be determined from the data. In these cases, the I/O operation finishes the job of choosing a coding system. Very often you will want to find out afterwards which coding system was chosen.

`buffer-file-coding-system` [Variable]

This buffer-local variable records the coding system used for saving the buffer and for writing part of the buffer with `write-region`. If the text to be written cannot be safely encoded using the coding system specified by this variable, these operations select an alternative encoding by calling the function `select-safe-coding-system` (see Section 34.10.4 [User-Chosen Coding Systems], page 985). If selecting a different

encoding requires to ask the user to specify a coding system, `buffer-file-coding-system` is updated to the newly selected coding system.

`buffer-file-coding-system` does *not* affect sending text to a subprocess.

`save-buffer-coding-system` [Variable]
This variable specifies the coding system for saving the buffer (by overriding `buffer-file-coding-system`). Note that it is not used for `write-region`.

When a command to save the buffer starts out to use `buffer-file-coding-system` (or `save-buffer-coding-system`), and that coding system cannot handle the actual text in the buffer, the command asks the user to choose another coding system (by calling `select-safe-coding-system`). After that happens, the command also updates `buffer-file-coding-system` to represent the coding system that the user specified.

`last-coding-system-used` [Variable]
I/O operations for files and subprocesses set this variable to the coding system name that was used. The explicit encoding and decoding functions (see Section 34.10.7 [Explicit Encoding], page 991) set it too.

Warning: Since receiving subprocess output sets this variable, it can change whenever Emacs waits; therefore, you should copy the value shortly after the function call that stores the value you are interested in.

The variable `selection-coding-system` specifies how to encode selections for the window system. See Section 30.20 [Window System Selections], page 842.

`file-name-coding-system` [Variable]
The variable `file-name-coding-system` specifies the coding system to use for encoding file names. Emacs encodes file names using that coding system for all file operations. If `file-name-coding-system` is `nil`, Emacs uses a default coding system determined by the selected language environment. In the default language environment, any non-ASCII characters in file names are not encoded specially; they appear in the file system using the internal Emacs representation.

Warning: if you change `file-name-coding-system` (or the language environment) in the middle of an Emacs session, problems can result if you have already visited files whose names were encoded using the earlier coding system and are handled differently under the new coding system. If you try to save one of these buffers under the visited file name, saving may use the wrong file name, or it may get an error. If such a problem happens, use `C-x C-w` to specify a new file name for that buffer.

On Windows 2000 and later, Emacs by default uses Unicode APIs to pass file names to the OS, so the value of `file-name-coding-system` is largely ignored. Lisp applications that need to encode or decode file names on the Lisp level should use `utf-8` coding-system when `system-type` is `windows-nt`; the conversion of UTF-8 encoded file names to the encoding appropriate for communicating with the OS is performed internally by Emacs.

34.10.3 Coding Systems in Lisp

Here are the Lisp facilities for working with coding systems:

`coding-system-list` **&optional** *base-only* [Function]
This function returns a list of all coding system names (symbols). If *base-only* is non-`nil`, the value includes only the base coding systems. Otherwise, it includes alias and variant coding systems as well.

`coding-system-p` *object* [Function]
This function returns `t` if *object* is a coding system name or `nil`.

`check-coding-system` *coding-system* [Function]
This function checks the validity of *coding-system*. If that is valid, it returns *coding-system*. If *coding-system* is `nil`, the function returns `nil`. For any other values, it signals an error whose `error-symbol` is `coding-system-error` (see Section 11.7.3.1 [Signaling Errors], page 180).

`coding-system-eol-type` *coding-system* [Function]
This function returns the type of end-of-line (a.k.a. *eol*) conversion used by *coding-system*. If *coding-system* specifies a certain eol conversion, the return value is an integer 0, 1, or 2, standing for `unix`, `dos`, and `mac`, respectively. If *coding-system* doesn't specify eol conversion explicitly, the return value is a vector of coding systems, each one with one of the possible eol conversion types, like this:

```
(coding-system-eol-type 'latin-1)
     ⇒ [latin-1-unix latin-1-dos latin-1-mac]
```

If this function returns a vector, Emacs will decide, as part of the text encoding or decoding process, what eol conversion to use. For decoding, the end-of-line format of the text is auto-detected, and the eol conversion is set to match it (e.g., DOS-style CRLF format will imply `dos` eol conversion). For encoding, the eol conversion is taken from the appropriate default coding system (e.g., default value of `buffer-file-coding-system` for `buffer-file-coding-system`), or from the default eol conversion appropriate for the underlying platform.

`coding-system-change-eol-conversion` *coding-system eol-type* [Function]
This function returns a coding system which is like *coding-system* except for its eol conversion, which is specified by `eol-type`. *eol-type* should be `unix`, `dos`, `mac`, or `nil`. If it is `nil`, the returned coding system determines the end-of-line conversion from the data.

eol-type may also be 0, 1 or 2, standing for `unix`, `dos` and `mac`, respectively.

`coding-system-change-text-conversion` *eol-coding text-coding* [Function]
This function returns a coding system which uses the end-of-line conversion of *eol-coding*, and the text conversion of *text-coding*. If *text-coding* is `nil`, it returns `undecided`, or one of its variants according to *eol-coding*.

`find-coding-systems-region` *from to* [Function]
This function returns a list of coding systems that could be used to encode a text between *from* and *to*. All coding systems in the list can safely encode any multibyte characters in that portion of the text.

If the text contains no multibyte characters, the function returns the list `(undecided)`.

`find-coding-systems-string` *string* [Function]
: This function returns a list of coding systems that could be used to encode the text of *string*. All coding systems in the list can safely encode any multibyte characters in *string*. If the text contains no multibyte characters, this returns the list `(undecided)`.

`find-coding-systems-for-charsets` *charsets* [Function]
: This function returns a list of coding systems that could be used to encode all the character sets in the list *charsets*.

`check-coding-systems-region` *start end coding-system-list* [Function]
: This function checks whether coding systems in the list `coding-system-list` can encode all the characters in the region between *start* and *end*. If all of the coding systems in the list can encode the specified text, the function returns `nil`. If some coding systems cannot encode some of the characters, the value is an alist, each element of which has the form `(`*`coding-system1 pos1 pos2 ...`*`)`, meaning that *coding-system1* cannot encode characters at buffer positions *pos1*, *pos2*, ...

  start may be a string, in which case *end* is ignored and the returned value references string indices instead of buffer positions.

`detect-coding-region` *start end* **&optional** *highest* [Function]
: This function chooses a plausible coding system for decoding the text from *start* to *end*. This text should be a byte sequence, i.e., unibyte text or multibyte text with only ASCII and eight-bit characters (see Section 34.10.7 [Explicit Encoding], page 991).

  Normally this function returns a list of coding systems that could handle decoding the text that was scanned. They are listed in order of decreasing priority. But if *highest* is non-`nil`, then the return value is just one coding system, the one that is highest in priority.

  If the region contains only ASCII characters except for such ISO-2022 control characters ISO-2022 as `ESC`, the value is `undecided` or `(undecided)`, or a variant specifying end-of-line conversion, if that can be deduced from the text.

  If the region contains null bytes, the value is `no-conversion`, even if the region contains text encoded in some coding system.

`detect-coding-string` *string* **&optional** *highest* [Function]
: This function is like `detect-coding-region` except that it operates on the contents of *string* instead of bytes in the buffer.

`inhibit-null-byte-detection` [Variable]
: If this variable has a non-`nil` value, null bytes are ignored when detecting the encoding of a region or a string. This allows the encoding of text that contains null bytes to be correctly detected, such as Info files with Index nodes.

`inhibit-iso-escape-detection` [Variable]
: If this variable has a non-`nil` value, ISO-2022 escape sequences are ignored when detecting the encoding of a region or a string. The result is that no text is ever detected as encoded in some ISO-2022 encoding, and all escape sequences become visible in a buffer. **Warning:** *Use this variable with extreme caution, because many files in the Emacs distribution use ISO-2022 encoding.*

`coding-system-charset-list` *coding-system* [Function]

This function returns the list of character sets (see Section 34.7 [Character Sets], page 976) supported by *coding-system*. Some coding systems that support too many character sets to list them all yield special values:

- If *coding-system* supports all Emacs characters, the value is `(emacs)`.
- If *coding-system* supports all Unicode characters, the value is `(unicode)`.
- If *coding-system* supports all ISO-2022 charsets, the value is `iso-2022`.
- If *coding-system* supports all the characters in the internal coding system used by Emacs version 21 (prior to the implementation of internal Unicode support), the value is `emacs-mule`.

See [Process Information], page 1098, in particular the description of the functions `process-coding-system` and `set-process-coding-system`, for how to examine or set the coding systems used for I/O to a subprocess.

34.10.4 User-Chosen Coding Systems

`select-safe-coding-system` *from to* **&optional** *default-coding-system accept-default-p file* [Function]

This function selects a coding system for encoding specified text, asking the user to choose if necessary. Normally the specified text is the text in the current buffer between *from* and *to*. If *from* is a string, the string specifies the text to encode, and *to* is ignored.

If the specified text includes raw bytes (see Section 34.1 [Text Representations], page 966), `select-safe-coding-system` suggests `raw-text` for its encoding.

If *default-coding-system* is non-`nil`, that is the first coding system to try; if that can handle the text, `select-safe-coding-system` returns that coding system. It can also be a list of coding systems; then the function tries each of them one by one. After trying all of them, it next tries the current buffer's value of `buffer-file-coding-system` (if it is not `undecided`), then the default value of `buffer-file-coding-system` and finally the user's most preferred coding system, which the user can set using the command `prefer-coding-system` (see Section "Recognizing Coding Systems" in *The GNU Emacs Manual*).

If one of those coding systems can safely encode all the specified text, `select-safe-coding-system` chooses it and returns it. Otherwise, it asks the user to choose from a list of coding systems which can encode all the text, and returns the user's choice.

default-coding-system can also be a list whose first element is `t` and whose other elements are coding systems. Then, if no coding system in the list can handle the text, `select-safe-coding-system` queries the user immediately, without trying any of the three alternatives described above. This is handy for checking only the coding systems in the list.

The optional argument *accept-default-p* determines whether a coding system selected without user interaction is acceptable. If it's omitted or `nil`, such a silent selection is always acceptable. If it is non-`nil`, it should be a function; `select-safe-coding-system` calls this function with one argument, the base coding system of the selected coding system. If the function returns `nil`, `select-safe-coding-system` rejects the

silently selected coding system, and asks the user to select a coding system from a list of possible candidates.

If the variable `select-safe-coding-system-accept-default-p` is non-`nil`, it should be a function taking a single argument. It is used in place of *accept-default-p*, overriding any value supplied for this argument.

As a final step, before returning the chosen coding system, `select-safe-coding-system` checks whether that coding system is consistent with what would be selected if the contents of the region were read from a file. (If not, this could lead to data corruption in a file subsequently re-visited and edited.) Normally, `select-safe-coding-system` uses `buffer-file-name` as the file for this purpose, but if *file* is non-`nil`, it uses that file instead (this can be relevant for `write-region` and similar functions). If it detects an apparent inconsistency, `select-safe-coding-system` queries the user before selecting the coding system.

`select-safe-coding-system-function` [Variable]

This variable names the function to be called to request the user to select a proper coding system for encoding text when the default coding system for an output operation cannot safely encode that text. The default value of this variable is `select-safe-coding-system`. Emacs primitives that write text to files, such as `write-region`, or send text to other processes, such as `process-send-region`, normally call the value of this variable, unless `coding-system-for-write` is bound to a non-`nil` value (see Section 34.10.6 [Specifying Coding Systems], page 989).

Here are two functions you can use to let the user specify a coding system, with completion. See Section 21.6 [Completion], page 396.

`read-coding-system` *prompt* **&optional** *default* [Function]

This function reads a coding system using the minibuffer, prompting with string *prompt*, and returns the coding system name as a symbol. If the user enters null input, *default* specifies which coding system to return. It should be a symbol or a string.

`read-non-nil-coding-system` *prompt* [Function]

This function reads a coding system using the minibuffer, prompting with string *prompt*, and returns the coding system name as a symbol. If the user tries to enter null input, it asks the user to try again. See Section 34.10 [Coding Systems], page 980.

34.10.5 Default Coding Systems

This section describes variables that specify the default coding system for certain files or when running certain subprograms, and the function that I/O operations use to access them.

The idea of these variables is that you set them once and for all to the defaults you want, and then do not change them again. To specify a particular coding system for a particular operation in a Lisp program, don't change these variables; instead, override them using `coding-system-for-read` and `coding-system-for-write` (see Section 34.10.6 [Specifying Coding Systems], page 989).

`auto-coding-regexp-alist` [User Option]

This variable is an alist of text patterns and corresponding coding systems. Each element has the form `(`*`regexp`* `.` *`coding-system`*`)`; a file whose first few kilobytes match *regexp* is decoded with *coding-system* when its contents are read into a buffer. The settings in this alist take priority over `coding:` tags in the files and the contents of `file-coding-system-alist` (see below). The default value is set so that Emacs automatically recognizes mail files in Babyl format and reads them with no code conversions.

`file-coding-system-alist` [User Option]

This variable is an alist that specifies the coding systems to use for reading and writing particular files. Each element has the form `(`*`pattern`* `.` *`coding`*`)`, where *pattern* is a regular expression that matches certain file names. The element applies to file names that match *pattern*.

The CDR of the element, *coding*, should be either a coding system, a cons cell containing two coding systems, or a function name (a symbol with a function definition). If *coding* is a coding system, that coding system is used for both reading the file and writing it. If *coding* is a cons cell containing two coding systems, its CAR specifies the coding system for decoding, and its CDR specifies the coding system for encoding.

If *coding* is a function name, the function should take one argument, a list of all arguments passed to `find-operation-coding-system`. It must return a coding system or a cons cell containing two coding systems. This value has the same meaning as described above.

If *coding* (or what returned by the above function) is `undecided`, the normal code-detection is performed.

`auto-coding-alist` [User Option]

This variable is an alist that specifies the coding systems to use for reading and writing particular files. Its form is like that of `file-coding-system-alist`, but, unlike the latter, this variable takes priority over any `coding:` tags in the file.

`process-coding-system-alist` [Variable]

This variable is an alist specifying which coding systems to use for a subprocess, depending on which program is running in the subprocess. It works like `file-coding-system-alist`, except that *pattern* is matched against the program name used to start the subprocess. The coding system or systems specified in this alist are used to initialize the coding systems used for I/O to the subprocess, but you can specify other coding systems later using `set-process-coding-system`.

Warning: Coding systems such as `undecided`, which determine the coding system from the data, do not work entirely reliably with asynchronous subprocess output. This is because Emacs handles asynchronous subprocess output in batches, as it arrives. If the coding system leaves the character code conversion unspecified, or leaves the end-of-line conversion unspecified, Emacs must try to detect the proper conversion from one batch at a time, and this does not always work.

Therefore, with an asynchronous subprocess, if at all possible, use a coding system which determines both the character code conversion and the end of line conversion—that is, one like `latin-1-unix`, rather than `undecided` or `latin-1`.

`network-coding-system-alist` [Variable]

This variable is an alist that specifies the coding system to use for network streams. It works much like `file-coding-system-alist`, with the difference that the *pattern* in an element may be either a port number or a regular expression. If it is a regular expression, it is matched against the network service name used to open the network stream.

`default-process-coding-system` [Variable]

This variable specifies the coding systems to use for subprocess (and network stream) input and output, when nothing else specifies what to do.

The value should be a cons cell of the form `(`*`input-coding`*` . `*`output-coding`*`)`. Here *input-coding* applies to input from the subprocess, and *output-coding* applies to output to it.

`auto-coding-functions` [User Option]

This variable holds a list of functions that try to determine a coding system for a file based on its undecoded contents.

Each function in this list should be written to look at text in the current buffer, but should not modify it in any way. The buffer will contain the text of parts of the file. Each function should take one argument, *size*, which tells it how many characters to look at, starting from point. If the function succeeds in determining a coding system for the file, it should return that coding system. Otherwise, it should return `nil`.

The functions in this list could be called either when the file is visited and Emacs wants to decode its contents, and/or when the file's buffer is about to be saved and Emacs wants to determine how to encode its contents.

If a file has a '`coding:`' tag, that takes precedence, so these functions won't be called.

`find-auto-coding` *filename size* [Function]

This function tries to determine a suitable coding system for *filename*. It examines the buffer visiting the named file, using the variables documented above in sequence, until it finds a match for one of the rules specified by these variables. It then returns a cons cell of the form `(`*`coding`*` . `*`source`*`)`, where *coding* is the coding system to use and *source* is a symbol, one of `auto-coding-alist`, `auto-coding-regexp-alist`, `:coding`, or `auto-coding-functions`, indicating which one supplied the matching rule. The value `:coding` means the coding system was specified by the `coding:` tag in the file (see Section "coding tag" in *The GNU Emacs Manual*). The order of looking for a matching rule is `auto-coding-alist` first, then `auto-coding-regexp-alist`, then the `coding:` tag, and lastly `auto-coding-functions`. If no matching rule was found, the function returns `nil`.

The second argument *size* is the size of text, in characters, following point. The function examines text only within *size* characters after point. Normally, the buffer should be positioned at the beginning when this function is called, because one of the places for the `coding:` tag is the first one or two lines of the file; in that case, *size* should be the size of the buffer.

`set-auto-coding` *filename size* [Function]
: This function returns a suitable coding system for file *filename*. It uses `find-auto-coding` to find the coding system. If no coding system could be determined, the function returns `nil`. The meaning of the argument *size* is like in `find-auto-coding`.

`find-operation-coding-system` *operation* **&rest** *arguments* [Function]
: This function returns the coding system to use (by default) for performing *operation* with *arguments*. The value has this form:

```
(decoding-system . encoding-system)
```

The first element, *decoding-system*, is the coding system to use for decoding (in case *operation* does decoding), and *encoding-system* is the coding system for encoding (in case *operation* does encoding).

The argument *operation* is a symbol; it should be one of `write-region`, `start-process`, `call-process`, `call-process-region`, `insert-file-contents`, or `open-network-stream`. These are the names of the Emacs I/O primitives that can do character code and eol conversion.

The remaining arguments should be the same arguments that might be given to the corresponding I/O primitive. Depending on the primitive, one of those arguments is selected as the *target*. For example, if *operation* does file I/O, whichever argument specifies the file name is the target. For subprocess primitives, the process name is the target. For `open-network-stream`, the target is the service name or port number.

Depending on *operation*, this function looks up the target in `file-coding-system-alist`, `process-coding-system-alist`, or `network-coding-system-alist`. If the target is found in the alist, `find-operation-coding-system` returns its association in the alist; otherwise it returns `nil`.

If *operation* is `insert-file-contents`, the argument corresponding to the target may be a cons cell of the form `(`*filename* `.` *buffer*`)`. In that case, *filename* is a file name to look up in `file-coding-system-alist`, and *buffer* is a buffer that contains the file's contents (not yet decoded). If `file-coding-system-alist` specifies a function to call for this file, and that function needs to examine the file's contents (as it usually does), it should examine the contents of *buffer* instead of reading the file.

34.10.6 Specifying a Coding System for One Operation

You can specify the coding system for a specific operation by binding the variables `coding-system-for-read` and/or `coding-system-for-write`.

`coding-system-for-read` [Variable]
: If this variable is non-`nil`, it specifies the coding system to use for reading a file, or for input from a synchronous subprocess.

It also applies to any asynchronous subprocess or network stream, but in a different way: the value of `coding-system-for-read` when you start the subprocess or open the network stream specifies the input decoding method for that subprocess or network stream. It remains in use for that subprocess or network stream unless and until overridden.

The right way to use this variable is to bind it with `let` for a specific I/O operation. Its global value is normally `nil`, and you should not globally set it to any other value. Here is an example of the right way to use the variable:

```
;; Read the file with no character code conversion.
(let ((coding-system-for-read 'no-conversion))
  (insert-file-contents filename))
```

When its value is non-`nil`, this variable takes precedence over all other methods of specifying a coding system to use for input, including `file-coding-system-alist`, `process-coding-system-alist` and `network-coding-system-alist`.

`coding-system-for-write` [Variable]

This works much like `coding-system-for-read`, except that it applies to output rather than input. It affects writing to files, as well as sending output to subprocesses and net connections. It also applies to encoding command-line arguments with which Emacs invokes subprocesses.

When a single operation does both input and output, as do `call-process-region` and `start-process`, both `coding-system-for-read` and `coding-system-for-write` affect it.

`coding-system-require-warning` [Variable]

Binding `coding-system-for-write` to a non-`nil` value prevents output primitives from calling the function specified by `select-safe-coding-system-function` (see Section 34.10.4 [User-Chosen Coding Systems], page 985). This is because `C-x RET c` (`universal-coding-system-argument`) works by binding `coding-system-for-write`, and Emacs should obey user selection. If a Lisp program binds `coding-system-for-write` to a value that might not be safe for encoding the text to be written, it can also bind `coding-system-require-warning` to a non-`nil` value, which will force the output primitives to check the encoding by calling the value of `select-safe-coding-system-function` even though `coding-system-for-write` is non-`nil`. Alternatively, call `select-safe-coding-system` explicitly before using the specified encoding.

`inhibit-eol-conversion` [User Option]

When this variable is non-`nil`, no end-of-line conversion is done, no matter which coding system is specified. This applies to all the Emacs I/O and subprocess primitives, and to the explicit encoding and decoding functions (see Section 34.10.7 [Explicit Encoding], page 991).

Sometimes, you need to prefer several coding systems for some operation, rather than fix a single one. Emacs lets you specify a priority order for using coding systems. This ordering affects the sorting of lists of coding systems returned by functions such as `find-coding-systems-region` (see Section 34.10.3 [Lisp and Coding Systems], page 982).

`coding-system-priority-list` **&optional** *highestp* [Function]

This function returns the list of coding systems in the order of their current priorities. Optional argument *highestp*, if non-`nil`, means return only the highest priority coding system.

`set-coding-system-priority` **&rest** *coding-systems* [Function]
This function puts *coding-systems* at the beginning of the priority list for coding systems, thus making their priority higher than all the rest.

`with-coding-priority` *coding-systems* **&rest** *body* [Macro]
This macro executes *body*, like `progn` does (see Section 11.1 [Sequencing], page 158), with *coding-systems* at the front of the priority list for coding systems. *coding-systems* should be a list of coding systems to prefer during execution of *body*.

34.10.7 Explicit Encoding and Decoding

All the operations that transfer text in and out of Emacs have the ability to use a coding system to encode or decode the text. You can also explicitly encode and decode text using the functions in this section.

The result of encoding, and the input to decoding, are not ordinary text. They logically consist of a series of byte values; that is, a series of ASCII and eight-bit characters. In unibyte buffers and strings, these characters have codes in the range 0 through #xFF (255). In a multibyte buffer or string, eight-bit characters have character codes higher than #xFF (see Section 34.1 [Text Representations], page 966), but Emacs transparently converts them to their single-byte values when you encode or decode such text.

The usual way to read a file into a buffer as a sequence of bytes, so you can decode the contents explicitly, is with `insert-file-contents-literally` (see Section 26.3 [Reading from Files], page 616); alternatively, specify a non-`nil` *rawfile* argument when visiting a file with `find-file-noselect`. These methods result in a unibyte buffer.

The usual way to use the byte sequence that results from explicitly encoding text is to copy it to a file or process—for example, to write it with `write-region` (see Section 26.4 [Writing to Files], page 618), and suppress encoding by binding `coding-system-for-write` to `no-conversion`.

Here are the functions to perform explicit encoding or decoding. The encoding functions produce sequences of bytes; the decoding functions are meant to operate on sequences of bytes. All of these functions discard text properties. They also set `last-coding-system-used` to the precise coding system they used.

`encode-coding-region` *start end coding-system* **&optional** *destination* [Command]
This command encodes the text from *start* to *end* according to coding system *coding-system*. Normally, the encoded text replaces the original text in the buffer, but the optional argument *destination* can change that. If *destination* is a buffer, the encoded text is inserted in that buffer after point (point does not move); if it is `t`, the command returns the encoded text as a unibyte string without inserting it.

If encoded text is inserted in some buffer, this command returns the length of the encoded text.

The result of encoding is logically a sequence of bytes, but the buffer remains multibyte if it was multibyte before, and any 8-bit bytes are converted to their multibyte representation (see Section 34.1 [Text Representations], page 966).

Do *not* use `undecided` for *coding-system* when encoding text, since that may lead to unexpected results. Instead, use `select-safe-coding-system` (see Section 34.10.4

[User-Chosen Coding Systems], page 985) to suggest a suitable encoding, if there's no obvious pertinent value for *coding-system*.

`encode-coding-string` *string coding-system* **&optional** *nocopy buffer* [Function]

This function encodes the text in *string* according to coding system *coding-system*. It returns a new string containing the encoded text, except when *nocopy* is non-`nil`, in which case the function may return *string* itself if the encoding operation is trivial. The result of encoding is a unibyte string.

`decode-coding-region` *start end coding-system* **&optional** *destination* [Command]

This command decodes the text from *start* to *end* according to coding system *coding-system*. To make explicit decoding useful, the text before decoding ought to be a sequence of byte values, but both multibyte and unibyte buffers are acceptable (in the multibyte case, the raw byte values should be represented as eight-bit characters). Normally, the decoded text replaces the original text in the buffer, but the optional argument *destination* can change that. If *destination* is a buffer, the decoded text is inserted in that buffer after point (point does not move); if it is `t`, the command returns the decoded text as a multibyte string without inserting it.

If decoded text is inserted in some buffer, this command returns the length of the decoded text. If that buffer is a unibyte buffer (see Section 34.4 [Selecting a Representation], page 970), the internal representation of the decoded text (see Section 34.1 [Text Representations], page 966) is inserted into the buffer as individual bytes.

This command puts a `charset` text property on the decoded text. The value of the property states the character set used to decode the original text.

This command detects the encoding of the text if necessary. If *coding-system* is `undecided`, the command detects the encoding of the text based on the byte sequences it finds in the text, and also detects the type of end-of-line convention used by the text (see Section 34.10.3 [Lisp and Coding Systems], page 982). If *coding-system* is `undecided-`*eol-type*, where *eol-type* is `unix`, `dos`, or `mac`, then the command detects only the encoding of the text. Any *coding-system* that doesn't specify *eol-type*, as in `utf-8`, causes the command to detect the end-of-line convention; specify the encoding completely, as in `utf-8-unix`, if the EOL convention used by the text is known in advance, to prevent any automatic detection.

`decode-coding-string` *string coding-system* **&optional** *nocopy buffer* [Function]

This function decodes the text in *string* according to *coding-system*. It returns a new string containing the decoded text, except when *nocopy* is non-`nil`, in which case the function may return *string* itself if the decoding operation is trivial. To make explicit decoding useful, the contents of *string* ought to be a unibyte string with a sequence of byte values, but a multibyte string is also acceptable (assuming it contains 8-bit bytes in their multibyte form).

This function detects the encoding of the string if needed, like `decode-coding-region` does.

If optional argument *buffer* specifies a buffer, the decoded text is inserted in that buffer after point (point does not move). In this case, the return value is the length of the decoded text. If that buffer is a unibyte buffer, the internal representation of the decoded text is inserted into it as individual bytes.

This function puts a `charset` text property on the decoded text. The value of the property states the character set used to decode the original text:

```
(decode-coding-string "Gr\374ss Gott" 'latin-1)
     ⇒ #("Grüss Gott" 0 9 (charset iso-8859-1))
```

`decode-coding-inserted-region` *from to filename* **&optional** *visit beg end replace* [Function]

This function decodes the text from *from* to *to* as if it were being read from file *filename* using `insert-file-contents` using the rest of the arguments provided.

The normal way to use this function is after reading text from a file without decoding, if you decide you would rather have decoded it. Instead of deleting the text and reading it again, this time with decoding, you can call this function.

34.10.8 Terminal I/O Encoding

Emacs can use coding systems to decode keyboard input and encode terminal output. This is useful for terminals that transmit or display text using a particular encoding, such as Latin-1. Emacs does not set `last-coding-system-used` when encoding or decoding terminal I/O.

`keyboard-coding-system` **&optional** *terminal* [Function]

This function returns the coding system used for decoding keyboard input from *terminal*. A value of `no-conversion` means no decoding is done. If *terminal* is omitted or `nil`, it means the selected frame's terminal. See Section 30.2 [Multiple Terminals], page 789.

`set-keyboard-coding-system` *coding-system* **&optional** *terminal* [Command]

This command specifies *coding-system* as the coding system to use for decoding keyboard input from *terminal*. If *coding-system* is `nil`, that means not to decode keyboard input. If *terminal* is a frame, it means that frame's terminal; if it is `nil`, that means the currently selected frame's terminal. See Section 30.2 [Multiple Terminals], page 789. Note that on modern MS-Windows systems Emacs always uses Unicode input when decoding keyboard input, so the encoding set by this command has no effect on Windows.

`terminal-coding-system` **&optional** *terminal* [Function]

This function returns the coding system that is in use for encoding terminal output from *terminal*. A value of `no-conversion` means no encoding is done. If *terminal* is a frame, it means that frame's terminal; if it is `nil`, that means the currently selected frame's terminal.

`set-terminal-coding-system` *coding-system* **&optional** *terminal* [Command]

This command specifies *coding-system* as the coding system to use for encoding terminal output from *terminal*. If *coding-system* is `nil`, that means not to encode terminal output. If *terminal* is a frame, it means that frame's terminal; if it is `nil`, that means the currently selected frame's terminal.

34.11 Input Methods

Input methods provide convenient ways of entering non-ASCII characters from the keyboard. Unlike coding systems, which translate non-ASCII characters to and from encodings meant to be read by programs, input methods provide human-friendly commands. (See Section "Input Methods" in *The GNU Emacs Manual*, for information on how users use input methods to enter text.) How to define input methods is not yet documented in this manual, but here we describe how to use them.

Each input method has a name, which is currently a string; in the future, symbols may also be usable as input method names.

`current-input-method` [Variable]
: This variable holds the name of the input method now active in the current buffer. (It automatically becomes local in each buffer when set in any fashion.) It is `nil` if no input method is active in the buffer now.

`default-input-method` [User Option]
: This variable holds the default input method for commands that choose an input method. Unlike `current-input-method`, this variable is normally global.

`set-input-method` *input-method* [Command]
: This command activates input method *input-method* for the current buffer. It also sets `default-input-method` to *input-method*. If *input-method* is `nil`, this command deactivates any input method for the current buffer.

`read-input-method-name` *prompt* **&optional** *default inhibit-null* [Function]
: This function reads an input method name with the minibuffer, prompting with *prompt*. If *default* is non-`nil`, that is returned by default, if the user enters empty input. However, if *inhibit-null* is non-`nil`, empty input signals an error.

  The returned value is a string.

`input-method-alist` [Variable]
: This variable defines all the supported input methods. Each element defines one input method, and should have the form:

```
(input-method language-env activate-func
 title description args...)
```

  Here *input-method* is the input method name, a string; *language-env* is another string, the name of the language environment this input method is recommended for. (That serves only for documentation purposes.)

  activate-func is a function to call to activate this method. The *args*, if any, are passed as arguments to *activate-func*. All told, the arguments to *activate-func* are *input-method* and the *args*.

  title is a string to display in the mode line while this method is active. *description* is a string describing this method and what it is good for.

The fundamental interface to input methods is through the variable `input-method-function`. See Section 22.8.2 [Reading One Event], page 463, and Section 22.8.4 [Invoking the Input Method], page 468.

34.12 Locales

In POSIX, locales control which language to use in language-related features. These Emacs variables control how Emacs interacts with these features.

`locale-coding-system` [Variable]
: This variable specifies the coding system to use for decoding system error messages and—on X Window system only—keyboard input, for sending batch output to the standard output and error streams, for encoding the format argument to `format-time-string`, and for decoding the return value of `format-time-string`.

`system-messages-locale` [Variable]
: This variable specifies the locale to use for generating system error messages. Changing the locale can cause messages to come out in a different language or in a different orthography. If the variable is `nil`, the locale is specified by environment variables in the usual POSIX fashion.

`system-time-locale` [Variable]
: This variable specifies the locale to use for formatting time values. Changing the locale can cause messages to appear according to the conventions of a different language. If the variable is `nil`, the locale is specified by environment variables in the usual POSIX fashion.

`locale-info` *item* [Function]
: This function returns locale data *item* for the current POSIX locale, if available. *item* should be one of these symbols:

  `codeset` Return the character set as a string (locale item `CODESET`).

  `days` Return a 7-element vector of day names (locale items `DAY_1` through `DAY_7`);

  `months` Return a 12-element vector of month names (locale items `MON_1` through `MON_12`).

  `paper` Return a list `(`*`width height`*`)` of 2 integers, for the default paper size measured in millimeters (locale items `_NL_PAPER_WIDTH` and `_NL_PAPER_HEIGHT`).

  If the system can't provide the requested information, or if *item* is not one of those symbols, the value is `nil`. All strings in the return value are decoded using `locale-coding-system`. See Section "Locales" in *The GNU Libc Manual*, for more information about locales and locale items.

35 Searching and Matching

GNU Emacs provides two ways to search through a buffer for specified text: exact string searches and regular expression searches. After a regular expression search, you can examine the *match data* to determine which text matched the whole regular expression or various portions of it.

The '`skip-chars...`' functions also perform a kind of searching. See Section 31.2.7 [Skipping Characters], page 865. To search for changes in character properties, see Section 33.19.3 [Property Search], page 924.

35.1 Searching for Strings

These are the primitive functions for searching through the text in a buffer. They are meant for use in programs, but you may call them interactively. If you do so, they prompt for the search string; the arguments *limit* and *noerror* are `nil`, and *repeat* is 1. For more details on interactive searching, see Section "Searching and Replacement" in *The GNU Emacs Manual*.

These search functions convert the search string to multibyte if the buffer is multibyte; they convert the search string to unibyte if the buffer is unibyte. See Section 34.1 [Text Representations], page 966.

`search-forward` *string* **&optional** *limit noerror count* [Command]

This function searches forward from point for an exact match for *string*. If successful, it sets point to the end of the occurrence found, and returns the new value of point. If no match is found, the value and side effects depend on *noerror* (see below).

In the following example, point is initially at the beginning of the line. Then `(search-forward "fox")` moves point after the last letter of '`fox`':

```
---------- Buffer: foo ----------
⋆The quick brown fox jumped over the lazy dog.
---------- Buffer: foo ----------

(search-forward "fox")
     ⇒ 20

---------- Buffer: foo ----------
The quick brown fox⋆ jumped over the lazy dog.
---------- Buffer: foo ----------
```

The argument *limit* specifies the bound to the search, and should be a position in the current buffer. No match extending after that position is accepted. If *limit* is omitted or `nil`, it defaults to the end of the accessible portion of the buffer.

What happens when the search fails depends on the value of *noerror*. If *noerror* is `nil`, a `search-failed` error is signaled. If *noerror* is `t`, `search-forward` returns `nil` and does nothing. If *noerror* is neither `nil` nor `t`, then `search-forward` moves point to the upper bound and returns `nil`.

The argument *noerror* only affects valid searches which fail to find a match. Invalid arguments cause errors regardless of *noerror*.

If *count* is a positive number *n*, the search is done *n* times; each successive search starts at the end of the previous match. If all these successive searches succeed, the function call succeeds, moving point and returning its new value. Otherwise the function call fails, with results depending on the value of *noerror*, as described above. If *count* is a negative number $-n$, the search is done *n* times in the opposite (backward) direction.

`search-backward` *string* **&optional** *limit noerror count* [Command]
This function searches backward from point for *string*. It is like `search-forward`, except that it searches backwards rather than forwards. Backward searches leave point at the beginning of the match.

`word-search-forward` *string* **&optional** *limit noerror count* [Command]
This function searches forward from point for a word match for *string*. If it finds a match, it sets point to the end of the match found, and returns the new value of point.

Word matching regards *string* as a sequence of words, disregarding punctuation that separates them. It searches the buffer for the same sequence of words. Each word must be distinct in the buffer (searching for the word '`ball`' does not match the word '`balls`'), but the details of punctuation and spacing are ignored (searching for '`ball boy`' does match '`ball. Boy!`').

In this example, point is initially at the beginning of the buffer; the search leaves it between the '`y`' and the '`!`'.

```
---------- Buffer: foo ----------
⋆He said "Please!  Find
the ball boy!"
---------- Buffer: foo ----------

(word-search-forward "Please find the ball, boy.")
     ⇒ 39

---------- Buffer: foo ----------
He said "Please!  Find
the ball boy⋆!"
---------- Buffer: foo ----------
```

If *limit* is non-`nil`, it must be a position in the current buffer; it specifies the upper bound to the search. The match found must not extend after that position.

If *noerror* is `nil`, then `word-search-forward` signals an error if the search fails. If *noerror* is `t`, then it returns `nil` instead of signaling an error. If *noerror* is neither `nil` nor `t`, it moves point to *limit* (or the end of the accessible portion of the buffer) and returns `nil`.

If *count* is a positive number, it specifies how many successive occurrences to search for. Point is positioned at the end of the last match. If *count* is a negative number, the search is backward and point is positioned at the beginning of the last match.

Internally, `word-search-forward` and related functions use the function `word-search-regexp` to convert *string* to a regular expression that ignores punctuation.

`word-search-forward-lax` *string* **&optional** *limit noerror count* [Command]
This command is identical to `word-search-forward`, except that the beginning or the end of *string* need not match a word boundary, unless *string* begins or ends in whitespace. For instance, searching for '`ball boy`' matches '`ball boyee`', but does not match '`balls boy`'.

`word-search-backward` *string* **&optional** *limit noerror count* [Command]
This function searches backward from point for a word match to *string*. This function is just like `word-search-forward` except that it searches backward and normally leaves point at the beginning of the match.

`word-search-backward-lax` *string* **&optional** *limit noerror count* [Command]
This command is identical to `word-search-backward`, except that the beginning or the end of *string* need not match a word boundary, unless *string* begins or ends in whitespace.

35.2 Searching and Case

By default, searches in Emacs ignore the case of the text they are searching through; if you specify searching for '`FOO`', then '`Foo`' or '`foo`' is also considered a match. This applies to regular expressions, too; thus, '`[aB]`' would match '`a`' or '`A`' or '`b`' or '`B`'.

If you do not want this feature, set the variable `case-fold-search` to `nil`. Then all letters must match exactly, including case. This is a buffer-local variable; altering the variable affects only the current buffer. (See Section 12.11.1 [Intro to Buffer-Local], page 207.) Alternatively, you may change the default value. In Lisp code, you will more typically use `let` to bind `case-fold-search` to the desired value.

Note that the user-level incremental search feature handles case distinctions differently. When the search string contains only lower case letters, the search ignores case, but when the search string contains one or more upper case letters, the search becomes case-sensitive. But this has nothing to do with the searching functions used in Lisp code. See Section "Incremental Search" in *The GNU Emacs Manual*.

`case-fold-search` [User Option]
This buffer-local variable determines whether searches should ignore case. If the variable is `nil` they do not ignore case; otherwise (and by default) they do ignore case.

`case-replace` [User Option]
This variable determines whether the higher-level replacement functions should preserve case. If the variable is `nil`, that means to use the replacement text verbatim. A non-`nil` value means to convert the case of the replacement text according to the text being replaced.

This variable is used by passing it as an argument to the function `replace-match`. See Section 35.6.1 [Replacing Match], page 1015.

35.3 Regular Expressions

A *regular expression*, or *regexp* for short, is a pattern that denotes a (possibly infinite) set of strings. Searching for matches for a regexp is a very powerful operation. This section explains how to write regexps; the following section says how to search for them.

For interactive development of regular expressions, you can use the *M-x re-builder* command. It provides a convenient interface for creating regular expressions, by giving immediate visual feedback in a separate buffer. As you edit the regexp, all its matches in the target buffer are highlighted. Each parenthesized sub-expression of the regexp is shown in a distinct face, which makes it easier to verify even very complex regexps.

Note that by default Emacs search ignores case (see Section 35.2 [Searching and Case], page 998). To enable case-sensitive regexp search and match, bind `case-fold-search` to `nil` around the code you want to be case-sensitive.

35.3.1 Syntax of Regular Expressions

Regular expressions have a syntax in which a few characters are special constructs and the rest are *ordinary*. An ordinary character is a simple regular expression that matches that character and nothing else. The special characters are '`.`', '`*`', '`+`', '`?`', '`[`', '`^`', '`$`', and '`\`'; no new special characters will be defined in the future. The character '`]`' is special if it ends a character alternative (see later). The character '`-`' is special inside a character alternative. A '`[:`' and balancing '`:]`' enclose a character class inside a character alternative. Any other character appearing in a regular expression is ordinary, unless a '`\`' precedes it.

For example, '`f`' is not a special character, so it is ordinary, and therefore '`f`' is a regular expression that matches the string '`f`' and no other string. (It does *not* match the string '`fg`', but it does match a *part* of that string.) Likewise, '`o`' is a regular expression that matches only '`o`'.

Any two regular expressions *a* and *b* can be concatenated. The result is a regular expression that matches a string if *a* matches some amount of the beginning of that string and *b* matches the rest of the string.

As a simple example, we can concatenate the regular expressions '`f`' and '`o`' to get the regular expression '`fo`', which matches only the string '`fo`'. Still trivial. To do something more powerful, you need to use one of the special regular expression constructs.

35.3.1.1 Special Characters in Regular Expressions

Here is a list of the characters that are special in a regular expression.

'`.`' (Period)
: is a special character that matches any single character except a newline. Using concatenation, we can make regular expressions like '`a.b`', which matches any three-character string that begins with '`a`' and ends with '`b`'.

'`*`'
: is not a construct by itself; it is a postfix operator that means to match the preceding regular expression repetitively as many times as possible. Thus, '`o*`' matches any number of '`o`'s (including no '`o`'s).

  '`*`' always applies to the *smallest* possible preceding expression. Thus, '`fo*`' has a repeating '`o`', not a repeating '`fo`'. It matches '`f`', '`fo`', '`foo`', and so on.

The matcher processes a '`*`' construct by matching, immediately, as many repetitions as can be found. Then it continues with the rest of the pattern. If that fails, backtracking occurs, discarding some of the matches of the '`*`'-modified construct in the hope that this will make it possible to match the rest of the pattern. For example, in matching '`ca*ar`' against the string '`caaar`', the '`a*`' first tries to match all three '`a`'s; but the rest of the pattern is '`ar`' and there is only '`r`' left to match, so this try fails. The next alternative is for '`a*`' to match only two '`a`'s. With this choice, the rest of the regexp matches successfully.

'`+`' is a postfix operator, similar to '`*`' except that it must match the preceding expression at least once. So, for example, '`ca+r`' matches the strings '`car`' and '`caaaar`' but not the string '`cr`', whereas '`ca*r`' matches all three strings.

'`?`' is a postfix operator, similar to '`*`' except that it must match the preceding expression either once or not at all. For example, '`ca?r`' matches '`car`' or '`cr`'; nothing else.

'`*?`', '`+?`', '`??`'
are *non-greedy* variants of the operators '`*`', '`+`' and '`?`'. Where those operators match the largest possible substring (consistent with matching the entire containing expression), the non-greedy variants match the smallest possible substring (consistent with matching the entire containing expression).

For example, the regular expression '`c[ad]*a`' when applied to the string '`cdaaada`' matches the whole string; but the regular expression '`c[ad]*?a`', applied to that same string, matches just '`cda`'. (The smallest possible match here for '`[ad]*?`' that permits the whole expression to match is '`d`'.)

'`[ ... ]`' is a *character alternative*, which begins with '`[`' and is terminated by '`]`'. In the simplest case, the characters between the two brackets are what this character alternative can match.

Thus, '`[ad]`' matches either one '`a`' or one '`d`', and '`[ad]*`' matches any string composed of just '`a`'s and '`d`'s (including the empty string). It follows that '`c[ad]*r`' matches '`cr`', '`car`', '`cdr`', '`caddaar`', etc.

You can also include character ranges in a character alternative, by writing the starting and ending characters with a '`-`' between them. Thus, '`[a-z]`' matches any lower-case ASCII letter. Ranges may be intermixed freely with individual characters, as in '`[a-z$%.]`', which matches any lower case ASCII letter or '`$`', '`%`' or period. However, the ending character of one range should not be the starting point of another one; for example, '`[a-m-z]`' should be avoided.

A character alternative can also specify named character classes (see Section 35.3.1.2 [Char Classes], page 1003). For example, '`[[:ascii:]]`' matches any ASCII character. Using a character class is equivalent to mentioning each of the characters in that class; but the latter is not feasible in practice, since some classes include thousands of different characters. A character class should not appear as the lower or upper bound of a range.

The usual regexp special characters are not special inside a character alternative. A completely different set of characters is special: '`]`', '`-`' and '`^`'. To include '`]`' in a character alternative, put it at the beginning. To include '`^`',

put it anywhere but at the beginning. To include '`-`', put it at the end. Thus, '`[]^-]`' matches all three of these special characters. You cannot use '`\`' to escape these three characters, since '`\`' is not special here.

The following aspects of ranges are specific to Emacs, in that POSIX allows but does not require this behavior and programs other than Emacs may behave differently:

1. If `case-fold-search` is non-`nil`, '`[a-z]`' also matches upper-case letters.
2. A range is not affected by the locale's collation sequence: it always represents the set of characters with codepoints ranging between those of its bounds, so that '`[a-z]`' matches only ASCII letters, even outside the C or POSIX locale.
3. If the lower bound of a range is greater than its upper bound, the range is empty and represents no characters. Thus, '`[z-a]`' always fails to match, and '`[^z-a]`' matches any character, including newline. However, a reversed range should always be from the letter '`z`' to the letter '`a`' to make it clear that it is not a typo; for example, '`[+-*/]`' should be avoided, because it matches only '`/`' rather than the likely-intended four characters.
4. If the end points of a range are raw 8-bit bytes (see Section 34.1 [Text Representations], page 966), or if the range start is ASCII and the end is a raw byte (as in '`[a-\377]`'), the range will match only ASCII characters and raw 8-bit bytes, but not non-ASCII characters. This feature is intended for searching text in unibyte buffers and strings.

Some kinds of character alternatives are not the best style even though they have a well-defined meaning in Emacs. They include:

1. Although a range's bound can be almost any character, it is better style to stay within natural sequences of ASCII letters and digits because most people have not memorized character code tables. For example, '`[.-9]`' is less clear than '`[./0-9]`', and '`[`-~]`' is less clear than '`[`a-z{|}~]`'. Unicode character escapes can help here; for example, for most programmers '`[--]`' is less clear than '`[\u0E01-\u0E3A\u0E3F-\u0E5B]`'.
2. Although a character alternative can include duplicates, it is better style to avoid them. For example, '`[XYa-yYb-zX]`' is less clear than '`[XYa-z]`'.
3. Although a range can denote just one, two, or three characters, it is simpler to list the characters. For example, '`[a-a0]`' is less clear than '`[a0]`', '`[i-j]`' is less clear than '`[ij]`', and '`[i-k]`' is less clear than '`[ijk]`'.
4. Although a '`-`' can appear at the beginning of a character alternative or as the upper bound of a range, it is better style to put '`-`' by itself at the end of a character alternative. For example, although '`[-a-z]`' is valid, '`[a-z-]`' is better style; and although '`[*--]`' is valid, '`[*+,-]`' is clearer.

'`[^ ... ]`' '`[^`' begins a *complemented character alternative*. This matches any character except the ones specified. Thus, '`[^a-z0-9A-Z]`' matches all characters *except* ASCII letters and digits.

‘^’ is not special in a character alternative unless it is the first character. The character following the ‘^’ is treated as if it were first (in other words, ‘-’ and ‘]’ are not special there).

A complemented character alternative can match a newline, unless newline is mentioned as one of the characters not to match. This is in contrast to the handling of regexps in programs such as `grep`.

You can specify named character classes, just like in character alternatives. For instance, ‘`[^[:ascii:]]`’ matches any non-ASCII character. See Section 35.3.1.2 [Char Classes], page 1003.

‘^’ When matching a buffer, ‘^’ matches the empty string, but only at the beginning of a line in the text being matched (or the beginning of the accessible portion of the buffer). Otherwise it fails to match anything. Thus, ‘`^foo`’ matches a ‘`foo`’ that occurs at the beginning of a line.

When matching a string instead of a buffer, ‘^’ matches at the beginning of the string or after a newline character.

For historical compatibility reasons, ‘^’ can be used only at the beginning of the regular expression, or after ‘`\(`’, ‘`\(?:`’ or ‘`\|`’.

‘$’ is similar to ‘^’ but matches only at the end of a line (or the end of the accessible portion of the buffer). Thus, ‘`x+$`’ matches a string of one ‘`x`’ or more at the end of a line.

When matching a string instead of a buffer, ‘$’ matches at the end of the string or before a newline character.

For historical compatibility reasons, ‘$’ can be used only at the end of the regular expression, or before ‘`\)`’ or ‘`\|`’.

‘\’ has two functions: it quotes the special characters (including ‘\’), and it introduces additional special constructs.

Because ‘\’ quotes special characters, ‘`\$`’ is a regular expression that matches only ‘$’, and ‘`\[`’ is a regular expression that matches only ‘[’, and so on.

Note that ‘\’ also has special meaning in the read syntax of Lisp strings (see Section 2.4.8 [String Type], page 20), and must be quoted with ‘\’. For example, the regular expression that matches the ‘\’ character is ‘`\\`’. To write a Lisp string that contains the characters ‘\\’, Lisp syntax requires you to quote each ‘\’ with another ‘\’. Therefore, the read syntax for a regular expression matching ‘\’ is `"\\\\"`.

Please note: For historical compatibility, special characters are treated as ordinary ones if they are in contexts where their special meanings make no sense. For example, ‘`*foo`’ treats ‘*’ as ordinary since there is no preceding expression on which the ‘*’ can act. It is poor practice to depend on this behavior; quote the special character anyway, regardless of where it appears.

As a ‘\’ is not special inside a character alternative, it can never remove the special meaning of ‘-’, ‘^’ or ‘]’. You should not quote these characters when they have no special meaning. This would not clarify anything, since backslashes can legitimately precede these characters where they *have* special meaning, as in ‘`[^\]`’ (`"[^\\]"` for Lisp string syntax), which matches any single character except a backslash.

In practice, most ‘]’ that occur in regular expressions close a character alternative and hence are special. However, occasionally a regular expression may try to match a complex pattern of literal ‘[’ and ‘]’. In such situations, it sometimes may be necessary to carefully parse the regexp from the start to determine which square brackets enclose a character alternative. For example, ‘[^][]]’ consists of the complemented character alternative ‘[^][]’ (which matches any single character that is not a square bracket), followed by a literal ‘]’.

The exact rules are that at the beginning of a regexp, ‘[’ is special and ‘]’ not. This lasts until the first unquoted ‘[’, after which we are in a character alternative; ‘[’ is no longer special (except when it starts a character class) but ‘]’ is special, unless it immediately follows the special ‘[’ or that ‘[’ followed by a ‘^’. This lasts until the next special ‘]’ that does not end a character class. This ends the character alternative and restores the ordinary syntax of regular expressions; an unquoted ‘[’ is special again and a ‘]’ not.

35.3.1.2 Character Classes

Below is a table of the classes you can use in a character alternative, and what they mean. Note that the ‘[’ and ‘]’ characters that enclose the class name are part of the name, so a regular expression using these classes needs one more pair of brackets. For example, a regular expression matching a sequence of one or more letters and digits would be ‘[[:alnum:]]+’, not ‘[:alnum:]+’.

‘[:ascii:]’
: This matches any ASCII character (codes 0–127).

‘[:alnum:]’
: This matches any letter or digit. For multibyte characters, it matches characters whose Unicode ‘general-category’ property (see Section 34.6 [Character Properties], page 971) indicates they are alphabetic or decimal number characters.

‘[:alpha:]’
: This matches any letter. For multibyte characters, it matches characters whose Unicode ‘general-category’ property (see Section 34.6 [Character Properties], page 971) indicates they are alphabetic characters.

‘[:blank:]’
: This matches horizontal whitespace, as defined by Annex C of the Unicode Technical Standard #18. In particular, it matches spaces, tabs, and other characters whose Unicode ‘general-category’ property (see Section 34.6 [Character Properties], page 971) indicates they are spacing separators.

‘[:cntrl:]’
: This matches any character whose code is in the range 0–31.

‘[:digit:]’
: This matches ‘0’ through ‘9’. Thus, ‘[-+[:digit:]]’ matches any digit, as well as ‘+’ and ‘-’.

‘[:graph:]’
: This matches graphic characters—everything except spaces, ASCII and non-ASCII control characters, surrogates, and codepoints unassigned by Unicode,

as indicated by the Unicode '`general-category`' property (see Section 34.6 [Character Properties], page 971).

'`[:lower:]`'
: This matches any lower-case letter, as determined by the current case table (see Section 4.10 [Case Tables], page 74). If `case-fold-search` is non-`nil`, this also matches any upper-case letter. Note that a buffer can have its own local case table different from the default one.

'`[:multibyte:]`'
: This matches any multibyte character (see Section 34.1 [Text Representations], page 966).

'`[:nonascii:]`'
: This matches any non-ASCII character.

'`[:print:]`'
: This matches any printing character—either spaces or graphic characters matched by '`[:graph:]`'.

'`[:punct:]`'
: This matches any punctuation character. (At present, for multibyte characters, it matches anything that has non-word syntax, and thus its exact definition can vary from one major mode to another, since the syntax of a character depends on the major mode.)

'`[:space:]`'
: This matches any character that has whitespace syntax (see Section 36.2.1 [Syntax Class Table], page 1026). Note that the syntax of a character, and thus which characters are considered "whitespace", depends on the major mode.

'`[:unibyte:]`'
: This matches any unibyte character (see Section 34.1 [Text Representations], page 966).

'`[:upper:]`'
: This matches any upper-case letter, as determined by the current case table (see Section 4.10 [Case Tables], page 74). If `case-fold-search` is non-`nil`, this also matches any lower-case letter. Note that a buffer can have its own local case table different from the default one.

'`[:word:]`'
: This matches any character that has word syntax (see Section 36.2.1 [Syntax Class Table], page 1026). Note that the syntax of a character, and thus which characters are considered "word-constituent", depends on the major mode.

'`[:xdigit:]`'
: This matches the hexadecimal digits: '`0`' through '`9`', '`a`' through '`f`' and '`A`' through '`F`'.

35.3.1.3 Backslash Constructs in Regular Expressions

For the most part, '\' followed by any character matches only that character. However, there are several exceptions: certain sequences starting with '\' that have special meanings. Here is a table of the special '\' constructs.

'`\|`' specifies an alternative. Two regular expressions *a* and *b* with '`\|`' in between form an expression that matches anything that either *a* or *b* matches.

Thus, '`foo\|bar`' matches either '`foo`' or '`bar`' but no other string.

'`\|`' applies to the largest possible surrounding expressions. Only a surrounding '`\( ... \)`' grouping can limit the grouping power of '`\|`'.

If you need full backtracking capability to handle multiple uses of '`\|`', use the POSIX regular expression functions (see Section 35.5 [POSIX Regexps], page 1014).

'`\{`*m*`\}`' is a postfix operator that repeats the previous pattern exactly *m* times. Thus, '`x\{5\}`' matches the string '`xxxxx`' and nothing else. '`c[ad]\{3\}r`' matches string such as '`caaar`', '`cdddr`', '`cadar`', and so on.

'`\{`*m*`,`*n*`\}`' is a more general postfix operator that specifies repetition with a minimum of *m* repeats and a maximum of *n* repeats. If *m* is omitted, the minimum is 0; if *n* is omitted, there is no maximum. For both forms, *m* and *n*, if specified, may be no larger than $2^{16} - 1$.

For example, '`c[ad]\{1,2\}r`' matches the strings '`car`', '`cdr`', '`caar`', '`cadr`', '`cdar`', and '`cddr`', and nothing else.
'`\{0,1\}`' or '`\{,1\}`' is equivalent to '`?`'.
'`\{0,\}`' or '`\{,\}`' is equivalent to '`*`'.
'`\{1,\}`' is equivalent to '`+`'.

'`\( ... \)`'
is a grouping construct that serves three purposes:

1. To enclose a set of '`\|`' alternatives for other operations. Thus, the regular expression '`\(foo\|bar\)x`' matches either '`foox`' or '`barx`'.
2. To enclose a complicated expression for the postfix operators '`*`', '`+`' and '`?`' to operate on. Thus, '`ba\(na\)*`' matches '`ba`', '`bana`', '`banana`', '`bananana`', etc., with any number (zero or more) of '`na`' strings.
3. To record a matched substring for future reference with '`\`*digit*' (see below).

This last application is not a consequence of the idea of a parenthetical grouping; it is a separate feature that was assigned as a second meaning to the same '`\( ... \)`' construct because, in practice, there was usually no conflict between the two meanings. But occasionally there is a conflict, and that led to the introduction of shy groups.

'`\(?: ... \)`'
is the *shy group* construct. A shy group serves the first two purposes of an ordinary group (controlling the nesting of other operators), but it does not get a number, so you cannot refer back to its value with '`\`*digit*'. Shy groups are

particularly useful for mechanically-constructed regular expressions, because they can be added automatically without altering the numbering of ordinary, non-shy groups.

Shy groups are also called *non-capturing* or *unnumbered groups*.

'`\(?num: ... \)`'
: is the *explicitly numbered group* construct. Normal groups get their number implicitly, based on their position, which can be inconvenient. This construct allows you to force a particular group number. There is no particular restriction on the numbering, e.g., you can have several groups with the same number in which case the last one to match (i.e., the rightmost match) will win. Implicitly numbered groups always get the smallest integer larger than the one of any previous group.

'`\digit`'
: matches the same text that matched the *digit*th occurrence of a grouping ('`\( ... \)`') construct.

  In other words, after the end of a group, the matcher remembers the beginning and end of the text matched by that group. Later on in the regular expression you can use '`\`' followed by *digit* to match that same text, whatever it may have been.

  The strings matching the first nine grouping constructs appearing in the entire regular expression passed to a search or matching function are assigned numbers 1 through 9 in the order that the open parentheses appear in the regular expression. So you can use '`\1`' through '`\9`' to refer to the text matched by the corresponding grouping constructs.

  For example, '`\(.*\)\1`' matches any newline-free string that is composed of two identical halves. The '`\(.*\)`' matches the first half, which may be anything, but the '`\1`' that follows must match the same exact text.

  If a '`\( ... \)`' construct matches more than once (which can happen, for instance, if it is followed by '`*`'), only the last match is recorded.

  If a particular grouping construct in the regular expression was never matched—for instance, if it appears inside of an alternative that wasn't used, or inside of a repetition that repeated zero times—then the corresponding '`\digit`' construct never matches anything. To use an artificial example, '`\(foo\(b*\)\|lose\)\2`' cannot match '`lose`': the second alternative inside the larger group matches it, but then '`\2`' is undefined and can't match anything. But it can match '`foobb`', because the first alternative matches '`foob`' and '`\2`' matches '`b`'.

'`\w`'
: matches any word-constituent character. The editor syntax table determines which characters these are. See Chapter 36 [Syntax Tables], page 1025.

'`\W`'
: matches any character that is not a word constituent.

'`\scode`'
: matches any character whose syntax is *code*. Here *code* is a character that represents a syntax code: thus, '`w`' for word constituent, '`-`' for whitespace, '`(`' for open parenthesis, etc. To represent whitespace syntax, use either '`-`' or a space character. See Section 36.2.1 [Syntax Class Table], page 1026, for a list of syntax codes and the characters that stand for them.

‘\S*code*’ matches any character whose syntax is not *code*.

‘\c*code*’ matches any character whose category is *code*. Here *code* is a character that represents a category: for example, in the standard category table, ‘c’ stands for Chinese characters and ‘g’ stands for Greek characters. You can see the list of all the currently defined categories with *M-x describe-categories* RET. You can also define your own categories in addition to the standard ones using the `define-category` function (see Section 36.8 [Categories], page 1039).

‘\C*code*’ matches any character whose category is not *code*.

The following regular expression constructs match the empty string—that is, they don't consume any characters—but whether they match depends on the context. For all, the beginning and end of the accessible portion of the buffer are treated as if they were the actual beginning and end of the buffer.

‘\`’ matches the empty string, but only at the beginning of the buffer or string being matched against.

‘\'’ matches the empty string, but only at the end of the buffer or string being matched against.

‘\=’ matches the empty string, but only at point. (This construct is not defined when matching against a string.)

‘\b’ matches the empty string, but only at the beginning or end of a word. Thus, ‘\bfoo\b’ matches any occurrence of ‘foo’ as a separate word. ‘\bballs?\b’ matches ‘ball’ or ‘balls’ as a separate word.

‘\b’ matches at the beginning or end of the buffer (or string) regardless of what text appears next to it.

‘\B’ matches the empty string, but *not* at the beginning or end of a word, nor at the beginning or end of the buffer (or string).

‘\<’ matches the empty string, but only at the beginning of a word. ‘\<’ matches at the beginning of the buffer (or string) only if a word-constituent character follows.

‘\>’ matches the empty string, but only at the end of a word. ‘\>’ matches at the end of the buffer (or string) only if the contents end with a word-constituent character.

‘_<’ matches the empty string, but only at the beginning of a symbol. A symbol is a sequence of one or more word or symbol constituent characters. ‘_<’ matches at the beginning of the buffer (or string) only if a symbol-constituent character follows.

‘_>’ matches the empty string, but only at the end of a symbol. ‘_>’ matches at the end of the buffer (or string) only if the contents end with a symbol-constituent character.

Not every string is a valid regular expression. For example, a string that ends inside a character alternative without a terminating ‘]’ is invalid, and so is a string that ends with a single ‘\’. If an invalid regular expression is passed to any of the search functions, an `invalid-regexp` error is signaled.

35.3.2 Complex Regexp Example

Here is a complicated regexp which was formerly used by Emacs to recognize the end of a sentence together with any whitespace that follows. (Nowadays Emacs uses a similar but more complex default regexp constructed by the function `sentence-end`. See Section 35.8 [Standard Regexps], page 1023.)

Below, we show first the regexp as a string in Lisp syntax (to distinguish spaces from tab characters), and then the result of evaluating it. The string constant begins and ends with a double-quote. '`\"`' stands for a double-quote as part of the string, '`\\`' for a backslash as part of the string, '`\t`' for a tab and '`\n`' for a newline.

```
"[.?!][]\"')}]*\\($\\| $\\|\t\\|  \\)[ \t\n]*"
     ⇒ "[.?!][]\"')}]*\\($\\| $\\|  \\|  \\)[
]*"
```

In the output, tab and newline appear as themselves.

This regular expression contains four parts in succession and can be deciphered as follows:

`[.?!]` The first part of the pattern is a character alternative that matches any one of three characters: period, question mark, and exclamation mark. The match must begin with one of these three characters. (This is one point where the new default regexp used by Emacs differs from the old. The new value also allows some non-ASCII characters that end a sentence without any following whitespace.)

`[]\"')}]*`
: The second part of the pattern matches any closing braces and quotation marks, zero or more of them, that may follow the period, question mark or exclamation mark. The `\"` is Lisp syntax for a double-quote in a string. The '`*`' at the end indicates that the immediately preceding regular expression (a character alternative, in this case) may be repeated zero or more times.

`\\($\\| $\\|\t\\|  \\)`
: The third part of the pattern matches the whitespace that follows the end of a sentence: the end of a line (optionally with a space), or a tab, or two spaces. The double backslashes mark the parentheses and vertical bars as regular expression syntax; the parentheses delimit a group and the vertical bars separate alternatives. The dollar sign is used to match the end of a line.

`[ \t\n]*` Finally, the last part of the pattern matches any additional whitespace beyond the minimum needed to end a sentence.

35.3.3 Regular Expression Functions

These functions operate on regular expressions.

`regexp-quote` *string* [Function]
: This function returns a regular expression whose only exact match is *string*. Using this regular expression in `looking-at` will succeed only if the next characters in the buffer are *string*; using it in a search function will succeed if the text being searched contains *string*. See Section 35.4 [Regexp Search], page 1011.

This allows you to request an exact string match or search when calling a function that wants a regular expression.

```
(regexp-quote "^The cat$")
     ⇒ "\\^The cat\\$"
```

One use of `regexp-quote` is to combine an exact string match with context described as a regular expression. For example, this searches for the string that is the value of *string*, surrounded by whitespace:

```
(re-search-forward
 (concat "\\s-" (regexp-quote string) "\\s-"))
```

The returned string may be *string* itself if it does not contain any special characters.

`regexp-opt` *strings* **&optional** *paren* [Function]

This function returns an efficient regular expression that will match any of the strings in the list *strings*. This is useful when you need to make matching or searching as fast as possible—for example, for Font Lock mode[1].

If *strings* is the empty list, the return value is a regexp that never matches anything.

The optional argument *paren* can be any of the following:

a string
: The resulting regexp is preceded by *paren* and followed by '`\)`', e.g. use '`"\\(?1:"`' to produce an explicitly numbered group.

`words`
: The resulting regexp is surrounded by '`\<\(`' and '`\)\>`'.

`symbols`
: The resulting regexp is surrounded by '`\_<\(`' and '`\)\_>`' (this is often appropriate when matching programming-language keywords and the like).

non-`nil`
: The resulting regexp is surrounded by '`\(`' and '`\)`'.

`nil`
: The resulting regexp is surrounded by '`\(?:`' and '`\)`', if it is necessary to ensure that a postfix operator appended to it will apply to the whole expression.

The returned regexp is ordered in such a way that it will always match the longest string possible.

Up to reordering, the resulting regexp of `regexp-opt` is equivalent to but usually more efficient than that of a simplified version:

```
(defun simplified-regexp-opt (strings &optional paren)
 (let ((parens
        (cond
         ((stringp paren)       (cons paren "\\)"))
         ((eq paren 'words)    '("\\<\\(" . "\\)\\>"))
         ((eq paren 'symbols) '("\\_<\\(" . "\\)\\_>"))
         ((null paren)          '("\\(?:" . "\\)"))
         (t                       '("\\(" . "\\)")))))
```

[1] Note that `regexp-opt` does not guarantee that its result is absolutely the most efficient form possible. A hand-tuned regular expression can sometimes be slightly more efficient, but is almost never worth the effort.

```
(concat (car parens)
        (mapconcat 'regexp-quote strings "\\|")
        (cdr parens))))
```

`regexp-opt-depth` *regexp* [Function]

This function returns the total number of grouping constructs (parenthesized expressions) in *regexp*. This does not include shy groups (see Section 35.3.1.3 [Regexp Backslash], page 1005).

`regexp-opt-charset` *chars* [Function]

This function returns a regular expression matching a character in the list of characters *chars*.

```
(regexp-opt-charset '(?a ?b ?c ?d ?e))
     ⇒ "[a-e]"
```

`regexp-unmatchable` [Variable]

This variable contains a regexp that is guaranteed not to match any string at all. It is particularly useful as default value for variables that may be set to a pattern that actually matches something.

35.3.4 Problems with Regular Expressions

The Emacs regexp implementation, like many of its kind, is generally robust but occasionally causes trouble in either of two ways: matching may run out of internal stack space and signal an error, and it can take a long time to complete. The advice below will make these symptoms less likely and help alleviate problems that do arise.

- Anchor regexps at the beginning of a line, string or buffer using zero-width assertions (‘`^`’ and ‘`\``’). This takes advantage of fast paths in the implementation and can avoid futile matching attempts. Other zero-width assertions may also bring benefits by causing a match to fail early.
- Avoid or-patterns in favor of character alternatives: write ‘`[ab]`’ instead of ‘`a\|b`’. Recall that ‘`\s-`’ and ‘`\sw`’ are equivalent to ‘`[[:space:]]`’ and ‘`[[:word:]]`’, respectively.
- Since the last branch of an or-pattern does not add a backtrack point on the stack, consider putting the most likely matched pattern last. For example, ‘`^\(?:a\|.b\)*c`’ will run out of stack if trying to match a very long string of ‘`a`’s, but the equivalent ‘`^\(?:.b\|a\)*c`’ will not.

  (It is a trade-off: successfully matched or-patterns run faster with the most frequently matched pattern first.)
- Try to ensure that any part of the text can only match in a single way. For example, ‘`a*a*`’ will match the same set of strings as ‘`a*`’, but the former can do so in many ways and will therefore cause slow backtracking if the match fails later on. Make or-pattern branches mutually exclusive if possible, so that matching will not go far into more than one branch before failing.

  Be especially careful with nested repetitions: they can easily result in very slow matching in the presence of ambiguities. For example, ‘`\(?:a*b*\)+c`’ will take a long time attempting to match even a moderately long string of ‘`a`’s before failing. The equivalent ‘`\(?:a\|b\)*c`’ is much faster, and ‘`[ab]*c`’ better still.

- Don't use capturing groups unless they are really needed; that is, use '`\(?:...\)`' instead of '`\(...\)`' for bracketing purposes.

If you run into regexp stack overflow despite following the above advice, don't be afraid of performing the matching in multiple function calls, each using a simpler regexp where backtracking can more easily be contained.

35.4 Regular Expression Searching

In GNU Emacs, you can search for the next match for a regular expression (see Section 35.3.1 [Syntax of Regexps], page 999) either incrementally or not. For incremental search commands, see Section "Regular Expression Search" in *The GNU Emacs Manual*. Here we describe only the search functions useful in programs. The principal one is `re-search-forward`.

These search functions convert the regular expression to multibyte if the buffer is multibyte; they convert the regular expression to unibyte if the buffer is unibyte. See Section 34.1 [Text Representations], page 966.

`re-search-forward` *regexp* **&optional** *limit noerror count* [Command]

This function searches forward in the current buffer for a string of text that is matched by the regular expression *regexp*. The function skips over any amount of text that is not matched by *regexp*, and leaves point at the end of the first match found. It returns the new value of point.

If *limit* is non-`nil`, it must be a position in the current buffer. It specifies the upper bound to the search. No match extending after that position is accepted. If *limit* is omitted or `nil`, it defaults to the end of the accessible portion of the buffer.

What `re-search-forward` does when the search fails depends on the value of *noerror*:

`nil` Signal a `search-failed` error.

`t` Do nothing and return `nil`.

anything else
Move point to *limit* (or the end of the accessible portion of the buffer) and return `nil`.

The argument *noerror* only affects valid searches which fail to find a match. Invalid arguments cause errors regardless of *noerror*.

If *count* is a positive number *n*, the search is done *n* times; each successive search starts at the end of the previous match. If all these successive searches succeed, the function call succeeds, moving point and returning its new value. Otherwise the function call fails, with results depending on the value of *noerror*, as described above. If *count* is a negative number $-n$, the search is done *n* times in the opposite (backward) direction.

In the following example, point is initially before the '`T`'. Evaluating the search call moves point to the end of that line (between the '`t`' of '`hat`' and the newline).

```
---------- Buffer: foo ----------
I read "⋆The cat in the hat
comes back" twice.
---------- Buffer: foo ----------
```

```
(re-search-forward "[a-z]+" nil t 5)
     ⇒ 27

---------- Buffer: foo ----------
I read "The cat in the hat*
comes back" twice.
---------- Buffer: foo ----------
```

`re-search-backward` *regexp* **&optional** *limit noerror count* [Command]

This function searches backward in the current buffer for a string of text that is matched by the regular expression *regexp*, leaving point at the beginning of the first text found.

This function is analogous to `re-search-forward`, but they are not simple mirror images. `re-search-forward` finds the match whose beginning is as close as possible to the starting point. If `re-search-backward` were a perfect mirror image, it would find the match whose end is as close as possible. However, in fact it finds the match whose beginning is as close as possible (and yet ends before the starting point). The reason for this is that matching a regular expression at a given spot always works from beginning to end, and starts at a specified beginning position.

A true mirror-image of `re-search-forward` would require a special feature for matching regular expressions from end to beginning. It's not worth the trouble of implementing that.

`string-match` *regexp string* **&optional** *start inhibit-modify* [Function]

This function returns the index of the start of the first match for the regular expression *regexp* in *string*, or `nil` if there is no match. If *start* is non-`nil`, the search starts at that index in *string*.

For example,

```
(string-match
 "quick" "The quick brown fox jumped quickly.")
     ⇒ 4
(string-match
 "quick" "The quick brown fox jumped quickly." 8)
     ⇒ 27
```

The index of the first character of the string is 0, the index of the second character is 1, and so on.

By default, if this function finds a match, the index of the first character beyond the match is available as `(match-end 0)`. See Section 35.6 [Match Data], page 1014. If *inhibit-modify* is non-`nil`, the match data isn't modified.

```
(string-match
 "quick" "The quick brown fox jumped quickly." 8)
     ⇒ 27

(match-end 0)
     ⇒ 32
```

`string-match-p` *regexp string* **&optional** *start* [Function]
This predicate function does what `string-match` does, but it avoids modifying the match data.

`looking-at` *regexp* **&optional** *inhibit-modify* [Function]
This function determines whether the text in the current buffer directly following point matches the regular expression *regexp*. "Directly following" means precisely that: the search is "anchored" and it can succeed only starting with the first character following point. The result is `t` if so, `nil` otherwise.

This function does not move point, but it does update the match data (if *inhibit-modify* is `nil` or missing, which is the default). See Section 35.6 [Match Data], page 1014. As a convenience, instead of using the *inhibit-modify* argument, you can use `looking-at-p`, described below.

In this example, point is located directly before the '`T`'. If it were anywhere else, the result would be `nil`.

```
---------- Buffer: foo ----------
I read "⋆The cat in the hat
comes back" twice.
---------- Buffer: foo ----------

(looking-at "The cat in the hat$")
     ⇒ t
```

`looking-back` *regexp limit* **&optional** *greedy* [Function]
This function returns `t` if *regexp* matches the text immediately before point (i.e., ending at point), and `nil` otherwise.

Because regular expression matching works only going forward, this is implemented by searching backwards from point for a match that ends at point. That can be quite slow if it has to search a long distance. You can bound the time required by specifying a non-`nil` value for *limit*, which says not to search before *limit*. In this case, the match that is found must begin at or after *limit*. Here's an example:

```
---------- Buffer: foo ----------
I read "⋆The cat in the hat
comes back" twice.
---------- Buffer: foo ----------

(looking-back "read \"" 3)
     ⇒ t
(looking-back "read \"" 4)
     ⇒ nil
```

If *greedy* is non-`nil`, this function extends the match backwards as far as possible, stopping when a single additional previous character cannot be part of a match for *regexp*. When the match is extended, its starting position is allowed to occur before *limit*.

As a general recommendation, try to avoid using `looking-back` wherever possible, since it is slow. For this reason, there are no plans to add a `looking-back-p` function.

`looking-at-p` *regexp* [Function]
: This predicate function works like `looking-at`, but without updating the match data.

`search-spaces-regexp` [Variable]
: If this variable is non-`nil`, it should be a regular expression that says how to search for whitespace. In that case, any group of spaces in a regular expression being searched for stands for use of this regular expression. However, spaces inside of constructs such as '`[...]`' and '`*`', '`+`', '`?`' are not affected by `search-spaces-regexp`.

  Since this variable affects all regular expression search and match constructs, you should bind it temporarily for as small as possible a part of the code, and only where the Lisp code affected by the binding performs searches whose regexp was produced from interactive user input. In other words, this variable should only be used to tell regexp search primitives how to interpret whitespace typed by the user.

35.5 POSIX Regular Expression Searching

The usual regular expression functions do backtracking when necessary to handle the '`\|`' and repetition constructs, but they continue this only until they find *some* match. Then they succeed and report the first match found.

This section describes alternative search functions which perform the full backtracking specified by the POSIX standard for regular expression matching. They continue backtracking until they have tried all possibilities and found all matches, so they can report the longest match, as required by POSIX. This is much slower, so use these functions only when you really need the longest match.

The POSIX search and match functions do not properly support the non-greedy repetition operators (see Section 35.3.1.1 [Regexp Special], page 999). This is because POSIX backtracking conflicts with the semantics of non-greedy repetition.

`posix-search-forward` *regexp* **&optional** *limit noerror count* [Command]
: This is like `re-search-forward` except that it performs the full backtracking specified by the POSIX standard for regular expression matching.

`posix-search-backward` *regexp* **&optional** *limit noerror count* [Command]
: This is like `re-search-backward` except that it performs the full backtracking specified by the POSIX standard for regular expression matching.

`posix-looking-at` *regexp* **&optional** *inhibit-modify* [Function]
: This is like `looking-at` except that it performs the full backtracking specified by the POSIX standard for regular expression matching.

`posix-string-match` *regexp string* **&optional** *start inhibit-modify* [Function]
: This is like `string-match` except that it performs the full backtracking specified by the POSIX standard for regular expression matching.

35.6 The Match Data

Emacs keeps track of the start and end positions of the segments of text found during a search; this is called the *match data*. Thanks to the match data, you can search for a

complex pattern, such as a date in a mail message, and then extract parts of the match under control of the pattern.

Because the match data normally describe the most recent search only, you must be careful not to do another search inadvertently between the search you wish to refer back to and the use of the match data. If you can't avoid another intervening search, you must save and restore the match data around it, to prevent it from being overwritten.

Notice that all functions are allowed to overwrite the match data unless they're explicitly documented not to do so. A consequence is that functions that are run implicitly in the background (see Section 42.11 [Timers], page 1286, and Section 42.12 [Idle Timers], page 1289) should likely save and restore the match data explicitly.

35.6.1 Replacing the Text that Matched

This function replaces all or part of the text matched by the last search. It works by means of the match data.

`replace-match` *replacement* **&optional** *fixedcase literal string subexp* [Function]

This function performs a replacement operation on a buffer or string.

If you did the last search in a buffer, you should omit the *string* argument or specify `nil` for it, and make sure that the current buffer is the one in which you performed the last search. Then this function edits the buffer, replacing the matched text with *replacement*. It leaves point at the end of the replacement text.

If you performed the last search on a string, pass the same string as *string*. Then this function returns a new string, in which the matched text is replaced by *replacement*.

If *fixedcase* is non-`nil`, then `replace-match` uses the replacement text without case conversion; otherwise, it converts the replacement text depending upon the capitalization of the text to be replaced. If the original text is all upper case, this converts the replacement text to upper case. If all words of the original text are capitalized, this capitalizes all the words of the replacement text. If all the words are one-letter and they are all upper case, they are treated as capitalized words rather than all-upper-case words.

If *literal* is non-`nil`, then *replacement* is inserted exactly as it is, the only alterations being case changes as needed. If it is `nil` (the default), then the character '`\`' is treated specially. If a '`\`' appears in *replacement*, then it must be part of one of the following sequences:

'`\&`' This stands for the entire text being replaced.

'`\`*n*', where *n* is a digit
: This stands for the text that matched the *n*th subexpression in the original regexp. Subexpressions are those expressions grouped inside '`\(...\)`'. If the *n*th subexpression never matched, an empty string is substituted.

'`\\`' This stands for a single '`\`' in the replacement text.

'`\?`' This stands for itself (for compatibility with `replace-regexp` and related commands; see Section "Regexp Replace" in *The GNU Emacs Manual*).

Any other character following '\' signals an error.

The substitutions performed by '\&' and '\n' occur after case conversion, if any. Therefore, the strings they substitute are never case-converted.

If *subexp* is non-`nil`, that says to replace just subexpression number *subexp* of the regexp that was matched, not the entire match. For example, after matching '`foo \(ba*r\)`', calling `replace-match` with 1 as *subexp* means to replace just the text that matched '`\(ba*r\)`'.

`match-substitute-replacement` *replacement* **&optional** *fixedcase literal string subexp* [Function]

This function returns the text that would be inserted into the buffer by `replace-match`, but without modifying the buffer. It is useful if you want to present the user with actual replacement result, with constructs like '\n' or '\&' substituted with matched groups. Arguments *replacement* and optional *fixedcase*, *literal*, *string* and *subexp* have the same meaning as for `replace-match`.

35.6.2 Simple Match Data Access

This section explains how to use the match data to find out what was matched by the last search or match operation, if it succeeded.

You can ask about the entire matching text, or about a particular parenthetical subexpression of a regular expression. The *count* argument in the functions below specifies which. If *count* is zero, you are asking about the entire match. If *count* is positive, it specifies which subexpression you want.

Recall that the subexpressions of a regular expression are those expressions grouped with escaped parentheses, '`\(...\)`'. The *count*th subexpression is found by counting occurrences of '`\(`' from the beginning of the whole regular expression. The first subexpression is numbered 1, the second 2, and so on. Only regular expressions can have subexpressions—after a simple string search, the only information available is about the entire match.

Every successful search sets the match data. Therefore, you should query the match data immediately after searching, before calling any other function that might perform another search. Alternatively, you may save and restore the match data (see Section 35.6.4 [Saving Match Data], page 1019) around the call to functions that could perform another search. Or use the functions that explicitly do not modify the match data; e.g., `string-match-p`.

A search which fails may or may not alter the match data. In the current implementation, it does not, but we may change it in the future. Don't try to rely on the value of the match data after a failing search.

`match-string` *count* **&optional** *in-string* [Function]

This function returns, as a string, the text matched in the last search or match operation. It returns the entire text if *count* is zero, or just the portion corresponding to the *count*th parenthetical subexpression, if *count* is positive.

If the last such operation was done against a string with `string-match`, then you should pass the same string as the argument *in-string*. After a buffer search or match, you should omit *in-string* or pass `nil` for it; but you should make sure that the current buffer when you call `match-string` is the one in which you did the searching or matching. Failure to follow this advice will lead to incorrect results.

The value is `nil` if *count* is out of range, or for a subexpression inside a '`\|`' alternative that wasn't used or a repetition that repeated zero times.

`match-string-no-properties` *count* **&optional** *in-string* [Function]
This function is like `match-string` except that the result has no text properties.

`match-beginning` *count* [Function]
If the last regular expression search found a match, this function returns the position of the start of the matching text or of a subexpression of it.

If *count* is zero, then the value is the position of the start of the entire match. Otherwise, *count* specifies a subexpression in the regular expression, and the value of the function is the starting position of the match for that subexpression.

The value is `nil` for a subexpression inside a '`\|`' alternative that wasn't used or a repetition that repeated zero times.

`match-end` *count* [Function]
This function is like `match-beginning` except that it returns the position of the end of the match, rather than the position of the beginning.

Here is an example of using the match data, with a comment showing the positions within the text:

```
(string-match "\\(qu\\)\\(ick\\)"
              "The quick fox jumped quickly.")
              ;0123456789
     ⇒ 4

(match-string 0 "The quick fox jumped quickly.")
     ⇒ "quick"
(match-string 1 "The quick fox jumped quickly.")
     ⇒ "qu"
(match-string 2 "The quick fox jumped quickly.")
     ⇒ "ick"

(match-beginning 1)       ; The beginning of the match
     ⇒ 4                  ;    with 'qu' is at index 4.

(match-beginning 2)       ; The beginning of the match
     ⇒ 6                  ;    with 'ick' is at index 6.

(match-end 1)             ; The end of the match
     ⇒ 6                  ;    with 'qu' is at index 6.

(match-end 2)             ; The end of the match
     ⇒ 9                  ;    with 'ick' is at index 9.
```

Here is another example. Point is initially located at the beginning of the line. Searching moves point to between the space and the word '`in`'. The beginning of the entire match

is at the 9th character of the buffer (‘T’), and the beginning of the match for the first subexpression is at the 13th character (‘c’).

```
(list
  (re-search-forward "The \\(cat \\)")
  (match-beginning 0)
  (match-beginning 1))
    ⇒ (17 9 13)

---------- Buffer: foo ----------
I read "The cat ⋆in the hat comes back" twice.
        ^   ^
        9  13
---------- Buffer: foo ----------
```

(In this case, the index returned is a buffer position; the first character of the buffer counts as 1.)

35.6.3 Accessing the Entire Match Data

The functions `match-data` and `set-match-data` read or write the entire match data, all at once.

`match-data` **&optional** *integers reuse reseat* [Function]

This function returns a list of positions (markers or integers) that record all the information on the text that the last search matched. Element zero is the position of the beginning of the match for the whole expression; element one is the position of the end of the match for the expression. The next two elements are the positions of the beginning and end of the match for the first subexpression, and so on. In general, element number $2n$ corresponds to `(match-beginning` *n*`)`; and element number $2n+1$ corresponds to `(match-end` *n*`)`.

Normally all the elements are markers or `nil`, but if *integers* is non-`nil`, that means to use integers instead of markers. (In that case, the buffer itself is appended as an additional element at the end of the list, to facilitate complete restoration of the match data.) If the last match was done on a string with `string-match`, then integers are always used, since markers can’t point into a string.

If *reuse* is non-`nil`, it should be a list. In that case, `match-data` stores the match data in *reuse*. That is, *reuse* is destructively modified. *reuse* does not need to have the right length. If it is not long enough to contain the match data, it is extended. If it is too long, the length of *reuse* stays the same, but the elements that were not used are set to `nil`. The purpose of this feature is to reduce the need for garbage collection.

If *reseat* is non-`nil`, all markers on the *reuse* list are reseated to point to nowhere.

As always, there must be no possibility of intervening searches between the call to a search function and the call to `match-data` that is intended to access the match data for that search.

```
(match-data)
     ⇒  (#<marker at 9 in foo>
         #<marker at 17 in foo>
         #<marker at 13 in foo>
         #<marker at 17 in foo>)
```

`set-match-data` *match-list* **&optional** *reseat* [Function]

This function sets the match data from the elements of *match-list*, which should be a list that was the value of a previous call to `match-data`. (More precisely, anything that has the same format will work.)

If *match-list* refers to a buffer that doesn't exist, you don't get an error; that sets the match data in a meaningless but harmless way.

If *reseat* is non-`nil`, all markers on the *match-list* list are reseated to point to nowhere.

`store-match-data` is a semi-obsolete alias for `set-match-data`.

35.6.4 Saving and Restoring the Match Data

When you call a function that may search, you may need to save and restore the match data around that call, if you want to preserve the match data from an earlier search for later use. Here is an example that shows the problem that arises if you fail to save the match data:

```
(re-search-forward "The \\(cat \\)")
     ⇒ 48
(foo)                   ; foo does more searching.
(match-end 0)
     ⇒ 61               ; Unexpected result—not 48!
```

You can save and restore the match data with `save-match-data`:

`save-match-data` *body...* [Macro]

This macro executes *body*, saving and restoring the match data around it. The return value is the value of the last form in *body*.

You could use `set-match-data` together with `match-data` to imitate the effect of the special form `save-match-data`. Here is how:

```
(let ((data (match-data)))
  (unwind-protect
      ...   ; Ok to change the original match data.
    (set-match-data data)))
```

Emacs automatically saves and restores the match data when it runs process filter functions (see Section 40.9.2 [Filter Functions], page 1104) and process sentinels (see Section 40.10 [Sentinels], page 1109).

35.7 Search and Replace

If you want to find all matches for a regexp in part of the buffer and replace them, the most flexible way is to write an explicit loop using `re-search-forward` and `replace-match`, like this:

```
(while (re-search-forward "foo[ \t]+bar" nil t)
```

```
    (replace-match "foobar"))
```

See Section 35.6.1 [Replacing the Text that Matched], page 1015, for a description of `replace-match`.

It may be more convenient to limit the replacements to a specific region. The function `replace-regexp-in-region` does that.

`replace-regexp-in-region` *regexp replacement* **&optional** *start end* [Function]
: This function replaces all the occurrences of *regexp* with *replacement* in the region of buffer text between *start* and *end*; *start* defaults to position of point, and *end* defaults to the last accessible position of the buffer. The search for *regexp* is case-sensitive, and *replacement* is inserted without changing its letter-case. The *replacement* string can use the same special elements starting with '\' as `replace-match` does. The function returns the number of replaced occurrences, or `nil` if *regexp* is not found. The function preserves the position of point.

```
(replace-regexp-in-region "foo[ \t]+bar" "foobar")
```

`replace-string-in-region` *string replacement* **&optional** *start end* [Function]
: This function works similarly to `replace-regexp-in-region`, but searches for, and replaces, literal *string*s instead of regular expressions.

Emacs also has special functions for replacing matches in a string.

`replace-regexp-in-string` *regexp rep string* **&optional** *fixedcase literal subexp start* [Function]
: This function copies *string* and searches it for matches for *regexp*, and replaces them with *rep*. It returns the modified copy. If *start* is non-`nil`, the search for matches starts at that index in *string*, and the returned value does not include the first *start* characters of *string*. To get the whole transformed string, concatenate the first *start* characters of *string* with the return value.

  This function uses `replace-match` to do the replacement, and it passes the optional arguments *fixedcase*, *literal* and *subexp* along to `replace-match`.

  Instead of a string, *rep* can be a function. In that case, `replace-regexp-in-string` calls *rep* for each match, passing the text of the match as its sole argument. It collects the value *rep* returns and passes that to `replace-match` as the replacement string. The match data at this point are the result of matching *regexp* against a substring of *string*.

`string-replace` *from-string to-string in-string* [Function]
: This function replaces all occurrences of *from-string* with *to-string* in *in-string* and returns the result. It may return one of its arguments unchanged, a constant string or a new string. Case is significant, and text properties are ignored.

If you want to write a command along the lines of `query-replace`, you can use `perform-replace` to do the work.

`perform-replace` *from-string replacements query-flag regexp-flag delimited-flag* **&optional** *repeat-count map start end backward region-noncontiguous-p* [Function]
: This function is the guts of `query-replace` and related commands. It searches for occurrences of *from-string* in the text between positions *start* and *end* and replaces

some or all of them. If *start* is `nil` (or omitted), point is used instead, and the end of the buffer's accessible portion is used for *end*. (If the optional argument *backward* is non-`nil`, the search starts at *end* and goes backward.)

If *query-flag* is `nil`, it replaces all occurrences; otherwise, it asks the user what to do about each one.

If *regexp-flag* is non-`nil`, then *from-string* is considered a regular expression; otherwise, it must match literally. If *delimited-flag* is non-`nil`, then only replacements surrounded by word boundaries are considered.

The argument *replacements* specifies what to replace occurrences with. If it is a string, that string is used. It can also be a list of strings, to be used in cyclic order.

If *replacements* is a cons cell, `(`*`function`*` . `*`data`*`)`, this means to call *function* after each match to get the replacement text. This function is called with two arguments: *data*, and the number of replacements already made.

If *repeat-count* is non-`nil`, it should be an integer. Then it specifies how many times to use each of the strings in the *replacements* list before advancing cyclically to the next one.

If *from-string* contains upper-case letters, then `perform-replace` binds `case-fold-search` to `nil`, and it uses the *replacements* without altering their case.

Normally, the keymap `query-replace-map` defines the possible user responses for queries. The argument *map*, if non-`nil`, specifies a keymap to use instead of `query-replace-map`.

Non-`nil` *region-noncontiguous-p* means that the region between *start* and *end* is composed of noncontiguous pieces. The most common example of this is a rectangular region, where the pieces are separated by newline characters.

This function uses one of two functions to search for the next occurrence of *from-string*. These functions are specified by the values of two variables: `replace-re-search-function` and `replace-search-function`. The former is called when the argument *regexp-flag* is non-`nil`, the latter when it is `nil`.

`query-replace-map` [Variable]

This variable holds a special keymap that defines the valid user responses for `perform-replace` and the commands that use it, as well as `y-or-n-p` and `map-y-or-n-p`. This map is unusual in two ways:

- The key bindings are not commands, just symbols that are meaningful to the functions that use this map.
- Prefix keys are not supported; each key binding must be for a single-event key sequence. This is because the functions don't use `read-key-sequence` to get the input; instead, they read a single event and look it up "by hand".

Here are the meaningful bindings for `query-replace-map`. Several of them are meaningful only for `query-replace` and friends.

`act` Do take the action being considered—in other words, "yes".

`skip` Do not take action for this question—in other words, "no".

`exit` Answer this question "no", and give up on the entire series of questions, assuming that the answers will be "no".

`exit-prefix`
Like `exit`, but add the key that was pressed to `unread-command-events` (see Section 22.8.6 [Event Input Misc], page 469).

`act-and-exit`
Answer this question "yes", and give up on the entire series of questions, assuming that subsequent answers will be "no".

`act-and-show`
Answer this question "yes", but show the results—don't advance yet to the next question.

`automatic`
Answer this question and all subsequent questions in the series with "yes", without further user interaction.

`backup` Move back to the previous place that a question was asked about.

`undo` Undo last replacement and move back to the place where that replacement was performed.

`undo-all` Undo all replacements and move back to the place where the first replacement was performed.

`edit` Enter a recursive edit to deal with this question—instead of any other action that would normally be taken.

`edit-replacement`
Edit the replacement for this question in the minibuffer.

`delete-and-edit`
Delete the text being considered, then enter a recursive edit to replace it.

`recenter`
`scroll-up`
`scroll-down`
`scroll-other-window`
`scroll-other-window-down`
Perform the specified window scroll operation, then ask the same question again. Only `y-or-n-p` and related functions use this answer.

`quit` Perform a quit right away. Only `y-or-n-p` and related functions use this answer.

`help` Display some help, then ask again.

`multi-query-replace-map` [Variable]
This variable holds a keymap that extends `query-replace-map` by providing additional key bindings that are useful in multi-buffer replacements. The additional bindings are:

`automatic-all`
Answer this question and all subsequent questions in the series with "yes", without further user interaction, for all remaining buffers.

`exit-current`
: Answer this question "no", and give up on the entire series of questions for the current buffer. Continue to the next buffer in the sequence.

`replace-search-function` [Variable]
: This variable specifies a function that `perform-replace` calls to search for the next string to replace. Its default value is `search-forward`. Any other value should name a function of 3 arguments: the first 3 arguments of `search-forward` (see Section 35.1 [String Search], page 996).

`replace-re-search-function` [Variable]
: This variable specifies a function that `perform-replace` calls to search for the next regexp to replace. Its default value is `re-search-forward`. Any other value should name a function of 3 arguments: the first 3 arguments of `re-search-forward` (see Section 35.4 [Regexp Search], page 1011).

35.8 Standard Regular Expressions Used in Editing

This section describes some variables that hold regular expressions used for certain purposes in editing:

`page-delimiter` [User Option]
: This is the regular expression describing line-beginnings that separate pages. The default value is `"^\014"` (i.e., `"^^L"` or `"^\C-l"`); this matches a line that starts with a formfeed character.

The following two regular expressions should *not* assume the match always starts at the beginning of a line; they should not use '`^`' to anchor the match. Most often, the paragraph commands do check for a match only at the beginning of a line, which means that '`^`' would be superfluous. When there is a nonzero left margin, they accept matches that start after the left margin. In that case, a '`^`' would be incorrect. However, a '`^`' is harmless in modes where a left margin is never used.

`paragraph-separate` [User Option]
: This is the regular expression for recognizing the beginning of a line that separates paragraphs. (If you change this, you may have to change `paragraph-start` also.) The default value is `"[ \t\f]*$"`, which matches a line that consists entirely of spaces, tabs, and form feeds (after its left margin).

`paragraph-start` [User Option]
: This is the regular expression for recognizing the beginning of a line that starts *or* separates paragraphs. The default value is `"\f\\|[ \t]*$"`, which matches a line containing only whitespace or starting with a form feed (after its left margin).

`sentence-end` [User Option]
: If non-`nil`, the value should be a regular expression describing the end of a sentence, including the whitespace following the sentence. (All paragraph boundaries also end sentences, regardless.)

  If the value is `nil`, as it is by default, then the function `sentence-end` constructs the regexp. That is why you should always call the function `sentence-end` to obtain the regexp to be used to recognize the end of a sentence.

`sentence-end` [Function]

This function returns the value of the variable `sentence-end`, if non-`nil`. Otherwise it returns a default value based on the values of the variables `sentence-end-double-space` (see [Definition of sentence-end-double-space], page 904), `sentence-end-without-period`, and `sentence-end-without-space`.

36 Syntax Tables

A *syntax table* specifies the syntactic role of each character in a buffer. It can be used to determine where words, symbols, and other syntactic constructs begin and end. This information is used by many Emacs facilities, including Font Lock mode (see Section 24.6 [Font Lock Mode], page 564) and the various complex movement commands (see Section 31.2 [Motion], page 856).

36.1 Syntax Table Concepts

A syntax table is a data structure which can be used to look up the *syntax class* and other syntactic properties of each character. Syntax tables are used by Lisp programs for scanning and moving across text.

Internally, a syntax table is a char-table (see Section 6.6 [Char-Tables], page 119). The element at index *c* describes the character with code *c*; its value is a cons cell which specifies the syntax of the character in question. See Section 36.7 [Syntax Table Internals], page 1037, for details. However, instead of using `aset` and `aref` to modify and inspect syntax table contents, you should usually use the higher-level functions `char-syntax` and `modify-syntax-entry`, which are described in Section 36.3 [Syntax Table Functions], page 1030.

`syntax-table-p` *object* [Function]
: This function returns `t` if *object* is a syntax table.

Each buffer has its own major mode, and each major mode has its own idea of the syntax class of various characters. For example, in Lisp mode, the character '`;`' begins a comment, but in C mode, it terminates a statement. To support these variations, the syntax table is local to each buffer. Typically, each major mode has its own syntax table, which it installs in all buffers that use that mode. For example, the variable `emacs-lisp-mode-syntax-table` holds the syntax table used by Emacs Lisp mode, and `c-mode-syntax-table` holds the syntax table used by C mode. Changing a major mode's syntax table alters the syntax in all of that mode's buffers, as well as in any buffers subsequently put in that mode. Occasionally, several similar modes share one syntax table. See Section 24.2.9 [Example Major Modes], page 543, for an example of how to set up a syntax table.

A syntax table can *inherit* from another syntax table, which is called its *parent syntax table*. A syntax table can leave the syntax class of some characters unspecified, by giving them the "inherit" syntax class; such a character then acquires the syntax class specified by the parent syntax table (see Section 36.2.1 [Syntax Class Table], page 1026). Emacs defines a *standard syntax table*, which is the default parent syntax table, and is also the syntax table used by Fundamental mode.

`standard-syntax-table` [Function]
: This function returns the standard syntax table, which is the syntax table used in Fundamental mode.

Syntax tables are not used by the Emacs Lisp reader, which has its own built-in syntactic rules which cannot be changed. (Some Lisp systems provide ways to redefine the read syntax, but we decided to leave this feature out of Emacs Lisp for simplicity.)

36.2 Syntax Descriptors

The *syntax class* of a character describes its syntactic role. Each syntax table specifies the syntax class of each character. There is no necessary relationship between the class of a character in one syntax table and its class in any other table.

Each syntax class is designated by a mnemonic character, which serves as the name of the class when you need to specify a class. Usually, this designator character is one that is often assigned that class; however, its meaning as a designator is unvarying and independent of what syntax that character currently has. Thus, '\' as a designator character always stands for escape character syntax, regardless of whether the '\' character actually has that syntax in the current syntax table.

A *syntax descriptor* is a Lisp string that describes the syntax class and other syntactic properties of a character. When you want to modify the syntax of a character, that is done by calling the function `modify-syntax-entry` and passing a syntax descriptor as one of its arguments (see Section 36.3 [Syntax Table Functions], page 1030).

The first character in a syntax descriptor must be a syntax class designator character. The second character, if present, specifies a matching character (e.g., in Lisp, the matching character for '`(`' is '`)`'); a space specifies that there is no matching character. Then come characters specifying additional syntax properties (see Section 36.2.2 [Syntax Flags], page 1029).

If no matching character or flags are needed, only one character (specifying the syntax class) is sufficient.

For example, the syntax descriptor for the character '`*`' in C mode is `". 23"` (i.e., punctuation, matching character slot unused, second character of a comment-starter, first character of a comment-ender), and the entry for '`/`' is '`. 14`' (i.e., punctuation, matching character slot unused, first character of a comment-starter, second character of a comment-ender).

Emacs also defines *raw syntax descriptors*, which are used to describe syntax classes at a lower level. See Section 36.7 [Syntax Table Internals], page 1037.

36.2.1 Table of Syntax Classes

Here is a table of syntax classes, the characters that designate them, their meanings, and examples of their use.

Whitespace characters: '` `' or '`-`'
: Characters that separate symbols and words from each other. Typically, whitespace characters have no other syntactic significance, and multiple whitespace characters are syntactically equivalent to a single one. Space, tab, and formfeed are classified as whitespace in almost all major modes.

  This syntax class can be designated by either '` `' or '`-`'. Both designators are equivalent.

Word constituents: '`w`'
: Parts of words in human languages. These are typically used in variable and command names in programs. All upper- and lower-case letters, and the digits, are typically word constituents.

Symbol constituents: '`_`'
: Extra characters used in variable and command names along with word constituents. Examples include the characters '`$&*+-_<>`' in Lisp mode, which may be part of a symbol name even though they are not part of English words. In standard C, the only non-word-constituent character that is valid in symbols is underscore ('`_`').

Punctuation characters: '`.`'
: Characters used as punctuation in a human language, or used in a programming language to separate symbols from one another. Some programming language modes, such as Emacs Lisp mode, have no characters in this class since the few characters that are not symbol or word constituents all have other uses. Other programming language modes, such as C mode, use punctuation syntax for operators.

Open parenthesis characters: '`(`'
Close parenthesis characters: '`)`'
: Characters used in dissimilar pairs to surround sentences or expressions. Such a grouping is begun with an open parenthesis character and terminated with a close. Each open parenthesis character matches a particular close parenthesis character, and vice versa. Normally, Emacs indicates momentarily the matching open parenthesis when you insert a close parenthesis. See Section 41.22 [Blinking], page 1246.

  In human languages, and in C code, the parenthesis pairs are '`()`', '`[]`', and '`{}`'. In Emacs Lisp, the delimiters for lists and vectors ('`()`' and '`[]`') are classified as parenthesis characters.

String quotes: '`"`'
: Characters used to delimit string constants. The same string quote character appears at the beginning and the end of a string. Such quoted strings do not nest.

  The parsing facilities of Emacs consider a string as a single token. The usual syntactic meanings of the characters in the string are suppressed.

  The Lisp modes have two string quote characters: double-quote ('`"`') and vertical bar ('`|`'). '`|`' is not used in Emacs Lisp, but it is used in Common Lisp. C also has two string quote characters: double-quote for strings, and apostrophe ('`'`') for character constants.

  Human text has no string quote characters. We do not want quotation marks to turn off the usual syntactic properties of other characters in the quotation.

Escape-syntax characters: '`\`'
: Characters that start an escape sequence, such as is used in string and character constants. The character '`\`' belongs to this class in both C and Lisp. (In C, it is used thus only inside strings, but it turns out to cause no trouble to treat it this way throughout C code.)

  Characters in this class count as part of words if `words-include-escapes` is non-`nil`. See Section 31.2.2 [Word Motion], page 857.

Character quotes: '`/`'
: Characters used to quote the following character so that it loses its normal syntactic meaning. This differs from an escape character in that only the character immediately following is ever affected.

  Characters in this class count as part of words if `words-include-escapes` is non-`nil`. See Section 31.2.2 [Word Motion], page 857.

  This class is used for backslash in TEX mode.

Paired delimiters: '`$`'
: Similar to string quote characters, except that the syntactic properties of the characters between the delimiters are not suppressed. Only TEX mode uses a paired delimiter presently—the '`$`' that both enters and leaves math mode.

Expression prefixes: '`'`'
: Characters used for syntactic operators that are considered as part of an expression if they appear next to one. In Lisp modes, these characters include the apostrophe, '`'`' (used for quoting), the comma, '`,`' (used in macros), and '`#`' (used in the read syntax for certain data types).

Comment starters: '`<`'
Comment enders: '`>`'
: Characters used in various languages to delimit comments. Human text has no comment characters. In Lisp, the semicolon ('`;`') starts a comment and a newline or formfeed ends one.

Inherit standard syntax: '`@`'
: This syntax class does not specify a particular syntax. It says to look in the parent syntax table to find the syntax of this character.

Generic comment delimiters: '`!`'
: (This syntax class is also known as "comment-fence".) Characters that start or end a special kind of comment. *Any* generic comment delimiter matches *any* generic comment delimiter, but they cannot match a comment starter or comment ender; generic comment delimiters can only match each other.

  This syntax class is primarily meant for use with the `syntax-table` text property (see Section 36.4 [Syntax Properties], page 1032). You can mark any range of characters as forming a comment, by giving the first and last characters of the range `syntax-table` properties identifying them as generic comment delimiters.

Generic string delimiters: '`|`'
: (This syntax class is also known as "string-fence".) Characters that start or end a string. This class differs from the string quote class in that *any* generic string delimiter can match any other generic string delimiter; but they do not match ordinary string quote characters.

  This syntax class is primarily meant for use with the `syntax-table` text property (see Section 36.4 [Syntax Properties], page 1032). You can mark any range of characters as forming a string constant, by giving the first and last characters of the range `syntax-table` properties identifying them as generic string delimiters.

36.2.2 Syntax Flags

In addition to the classes, entries for characters in a syntax table can specify flags. There are eight possible flags, represented by the characters '`1`', '`2`', '`3`', '`4`', '`b`', '`c`', '`n`', and '`p`'.

All the flags except '`p`' are used to describe comment delimiters. The digit flags are used for comment delimiters made up of 2 characters. They indicate that a character can *also* be part of a comment sequence, in addition to the syntactic properties associated with its character class. The flags are independent of the class and each other for the sake of characters such as '`*`' in C mode, which is a punctuation character, *and* the second character of a start-of-comment sequence ('`/*`'), *and* the first character of an end-of-comment sequence ('`*/`'). The flags '`b`', '`c`', and '`n`' are used to qualify the corresponding comment delimiter.

Here is a table of the possible flags for a character *c*, and what they mean:

- '`1`' means *c* is the start of a two-character comment-start sequence.
- '`2`' means *c* is the second character of such a sequence.
- '`3`' means *c* is the start of a two-character comment-end sequence.
- '`4`' means *c* is the second character of such a sequence.
- '`b`' means that *c* as a comment delimiter belongs to the alternative "b" comment style. For a two-character comment starter, this flag is only significant on the second char, and for a 2-character comment ender it is only significant on the first char.
- '`c`' means that *c* as a comment delimiter belongs to the alternative "c" comment style. For a two-character comment delimiter, '`c`' on either character makes it of style "c".
- '`n`' on a comment delimiter character specifies that this kind of comment can be nested. Inside such a comment, only comments of the same style will be recognized. For a two-character comment delimiter, '`n`' on either character makes it nestable.

  Emacs supports several comment styles simultaneously in any one syntax table. A comment style is a set of flags '`b`', '`c`', and '`n`', so there can be up to 8 different comment styles, each one named by the set of its flags. Each comment delimiter has a style and only matches comment delimiters of the same style. Thus if a comment starts with the comment-start sequence of style "bn", it will extend until the next matching comment-end sequence of style "bn". When the set of flags has neither flag '`b`' nor flag '`c`' set, the resulting style is called the "a" style.

  The appropriate comment syntax settings for C++ can be as follows:

  '`/`' '`124`'

  '`*`' '`23b`'

  newline '`>`'

  This defines four comment-delimiting sequences:

  '`/*`' This is a comment-start sequence for "b" style because the second character, '`*`', has the '`b`' flag.

  '`//`' This is a comment-start sequence for "a" style because the second character, '`/`', does not have the '`b`' flag.

  '`*/`' This is a comment-end sequence for "b" style because the first character, '`*`', has the '`b`' flag.

newline This is a comment-end sequence for "a" style, because the newline character does not have the '`b`' flag.

- '`p`' identifies an additional prefix character for Lisp syntax. These characters are treated as whitespace when they appear between expressions. When they appear within an expression, they are handled according to their usual syntax classes.

  The function `backward-prefix-chars` moves back over these characters, as well as over characters whose primary syntax class is prefix ('`'`'). See Section 36.5 [Motion and Syntax], page 1033.

36.3 Syntax Table Functions

In this section we describe functions for creating, accessing and altering syntax tables.

`make-syntax-table` **&optional** *table* [Function]
This function creates a new syntax table. If *table* is non-`nil`, the parent of the new syntax table is *table*; otherwise, the parent is the standard syntax table.

In the new syntax table, all characters are initially given the "inherit" ('`@`') syntax class, i.e., their syntax is inherited from the parent table (see Section 36.2.1 [Syntax Class Table], page 1026).

`copy-syntax-table` **&optional** *table* [Function]
This function constructs a copy of *table* and returns it. If *table* is omitted or `nil`, it returns a copy of the standard syntax table. Otherwise, an error is signaled if *table* is not a syntax table.

`modify-syntax-entry` *char syntax-descriptor* **&optional** *table* [Command]
This function sets the syntax entry for *char* according to *syntax-descriptor*. *char* must be a character, or a cons cell of the form `(`*min* `.` *max*`)`; in the latter case, the function sets the syntax entries for all characters in the range between *min* and *max*, inclusive.

The syntax is changed only for *table*, which defaults to the current buffer's syntax table, and not in any other syntax table.

The argument *syntax-descriptor* is a syntax descriptor, i.e., a string whose first character is a syntax class designator and whose second and subsequent characters optionally specify a matching character and syntax flags. See Section 36.2 [Syntax Descriptors], page 1026. An error is signaled if *syntax-descriptor* is not a valid syntax descriptor.

This function always returns `nil`. The old syntax information in the table for this character is discarded.

Examples:

```
;; Put the space character in class whitespace.
(modify-syntax-entry ?\s " ")
     ⇒ nil

;; Make '$' an open parenthesis character,
;;    with '^' as its matching close.
(modify-syntax-entry ?$ "(^")
     ⇒ nil
```

```
;; Make '^' a close parenthesis character,
;;   with '$' as its matching open.
(modify-syntax-entry ?^ ")$")
     ⇒ nil

;; Make '/' a punctuation character,
;;   the first character of a start-comment sequence,
;;   and the second character of an end-comment sequence.
;;   This is used in C mode.
(modify-syntax-entry ?/ ". 14")
     ⇒ nil
```

`char-syntax` *character* [Function]

This function returns the syntax class of *character*, represented by its designator character (see Section 36.2.1 [Syntax Class Table], page 1026). This returns *only* the class, not its matching character or syntax flags.

The following examples apply to C mode. (We use `string` to make it easier to see the character returned by `char-syntax`.)

```
;; Space characters have whitespace syntax class.
(string (char-syntax ?\s))
     ⇒ " "

;; Forward slash characters have punctuation syntax.
;; Note that this char-syntax call does not reveal
;; that it is also part of comment-start and -end sequences.
(string (char-syntax ?/))
     ⇒ "."

;; Open parenthesis characters have open parenthesis syntax.
;; Note that this char-syntax call does not reveal that
;; it has a matching character, ')'.
(string (char-syntax ?\())
     ⇒ "("
```

`set-syntax-table` *table* [Function]

This function makes *table* the syntax table for the current buffer. It returns *table*.

`syntax-table` [Function]

This function returns the current syntax table, which is the table for the current buffer.

`describe-syntax` **&optional** *buffer* [Command]

This command displays the contents of the syntax table of *buffer* (by default, the current buffer) in a help buffer.

`with-syntax-table` *table body...* [Macro]

This macro executes *body* using *table* as the current syntax table. It returns the value of the last form in *body*, after restoring the old current syntax table.

Since each buffer has its own current syntax table, we should make that more precise: `with-syntax-table` temporarily alters the current syntax table of whichever buffer is current at the time the macro execution starts. Other buffers are not affected.

36.4 Syntax Properties

When the syntax table is not flexible enough to specify the syntax of a language, you can override the syntax table for specific character occurrences in the buffer, by applying a `syntax-table` text property. See Section 33.19 [Text Properties], page 919, for how to apply text properties.

The valid values of `syntax-table` text property are:

syntax-table
: If the property value is a syntax table, that table is used instead of the current buffer's syntax table to determine the syntax for the underlying text character.

`(`*`syntax-code . matching-char`*`)`
: A cons cell of this format is a raw syntax descriptor (see Section 36.7 [Syntax Table Internals], page 1037), which directly specifies a syntax class for the underlying text character.

`nil`
: If the property is `nil`, the character's syntax is determined from the current syntax table in the usual way.

`parse-sexp-lookup-properties` [Variable]
: If this is non-`nil`, the syntax scanning functions, like `forward-sexp`, pay attention to `syntax-table` text properties. Otherwise they use only the current syntax table.

`syntax-propertize-function` [Variable]
: This variable, if non-`nil`, should store a function for applying `syntax-table` properties to a specified stretch of text. It is intended to be used by major modes to install a function which applies `syntax-table` properties in some mode-appropriate way.

  The function is called by `syntax-ppss` (see Section 36.6.2 [Position Parse], page 1035), and by Font Lock mode during syntactic fontification (see Section 24.6.8 [Syntactic Font Lock], page 577). It is called with two arguments, *start* and *end*, which are the starting and ending positions of the text on which it should act. It is allowed to arbitrarily move point within the region delimited by *start* and *end*; such motions don't need to use `save-excursion` (see Section 31.3 [Excursions], page 866). It is also allowed to call `syntax-ppss` on any position before *end*, but if a Lisp program calls `syntax-ppss` on some position and later modifies the buffer at some earlier position, then it is that program's responsibility to call `syntax-ppss-flush-cache` to flush the now obsolete info from the cache.

  Caution: When this variable is non-`nil`, Emacs removes `syntax-table` text properties arbitrarily and relies on `syntax-propertize-function` to reapply them. Thus if this facility is used at all, the function must apply **all** `syntax-table` text properties used by the major mode. In particular, Modes derived from a CC Mode mode must not use this variable, since CC Mode uses other means to apply and remove these text properties.

`syntax-propertize-extend-region-functions` [Variable]
: This abnormal hook is run by the syntax parsing code prior to calling `syntax-propertize-function`. Its role is to help locate safe starting and ending buffer positions for passing to `syntax-propertize-function`. For example, a major mode can add a function to this hook to identify multi-line syntactic constructs, and ensure that the boundaries do not fall in the middle of one.

  Each function in this hook should accept two arguments, *start* and *end*. It should return either a cons cell of two adjusted buffer positions, `(`*`new-start`*` . `*`new-end`*`)`, or `nil` if no adjustment is necessary. The hook functions are run in turn, repeatedly, until they all return `nil`.

36.5 Motion and Syntax

This section describes functions for moving across characters that have certain syntax classes.

`skip-syntax-forward` *syntaxes* **&optional** *limit* [Function]
: This function moves point forward across characters having syntax classes mentioned in *syntaxes* (a string of syntax class characters). It stops when it encounters the end of the buffer, or position *limit* (if specified), or a character it is not supposed to skip.

  If *syntaxes* starts with '`^`', then the function skips characters whose syntax is *not* in *syntaxes*.

  The return value is the distance traveled, which is a nonnegative integer.

`skip-syntax-backward` *syntaxes* **&optional** *limit* [Function]
: This function moves point backward across characters whose syntax classes are mentioned in *syntaxes*. It stops when it encounters the beginning of the buffer, or position *limit* (if specified), or a character it is not supposed to skip.

  If *syntaxes* starts with '`^`', then the function skips characters whose syntax is *not* in *syntaxes*.

  The return value indicates the distance traveled. It is an integer that is zero or less.

`backward-prefix-chars` [Function]
: This function moves point backward over any number of characters with expression prefix syntax. This includes both characters in the expression prefix syntax class, and characters with the '`p`' flag.

36.6 Parsing Expressions

This section describes functions for parsing and scanning balanced expressions. We will refer to such expressions as *sexps*, following the terminology of Lisp, even though these functions can act on languages other than Lisp. Basically, a sexp is either a balanced parenthetical grouping, a string, or a symbol (i.e., a sequence of characters whose syntax is either word constituent or symbol constituent). However, characters in the expression prefix syntax class (see Section 36.2.1 [Syntax Class Table], page 1026) are treated as part of the sexp if they appear next to it.

The syntax table controls the interpretation of characters, so these functions can be used for Lisp expressions when in Lisp mode and for C expressions when in C mode. See

Section 31.2.6 [List Motion], page 863, for convenient higher-level functions for moving over balanced expressions.

A character's syntax controls how it changes the state of the parser, rather than describing the state itself. For example, a string delimiter character toggles the parser state between in-string and in-code, but the syntax of characters does not directly say whether they are inside a string. For example (note that 15 is the syntax code for generic string delimiters),

```
(put-text-property 1 9 'syntax-table '(15 . nil))
```

does not tell Emacs that the first eight chars of the current buffer are a string, but rather that they are all string delimiters. As a result, Emacs treats them as four consecutive empty string constants.

36.6.1 Motion Commands Based on Parsing

This section describes simple point-motion functions that operate based on parsing expressions.

`scan-lists` *from count depth* [Function]

This function scans forward *count* balanced parenthetical groupings from position *from*. It returns the position where the scan stops. If *count* is negative, the scan moves backwards.

If *depth* is nonzero, treat the starting position as being *depth* parentheses deep. The scanner moves forward or backward through the buffer until the depth changes to zero *count* times. Hence, a positive value for *depth* has the effect of moving out *depth* levels of parenthesis from the starting position, while a negative *depth* has the effect of moving deeper by -*depth* levels of parenthesis.

Scanning ignores comments if `parse-sexp-ignore-comments` is non-`nil`.

If the scan reaches the beginning or end of the accessible part of the buffer before it has scanned over *count* parenthetical groupings, the return value is `nil` if the depth at that point is zero; if the depth is non-zero, a `scan-error` error is signaled.

`scan-sexps` *from count* [Function]

This function scans forward *count* sexps from position *from*. It returns the position where the scan stops. If *count* is negative, the scan moves backwards.

Scanning ignores comments if `parse-sexp-ignore-comments` is non-`nil`.

If the scan reaches the beginning or end of (the accessible part of) the buffer while in the middle of a parenthetical grouping, an error is signaled. If it reaches the beginning or end between groupings but before count is used up, `nil` is returned.

`forward-comment` *count* [Function]

This function moves point forward across *count* complete comments (that is, including the starting delimiter and the terminating delimiter if any), plus any whitespace encountered on the way. It moves backward if *count* is negative. If it encounters anything other than a comment or whitespace, it stops, leaving point at the place where it stopped. This includes (for instance) finding the end of a comment when moving forward and expecting the beginning of one. The function also stops immediately after moving over the specified number of complete comments. If *count* comments

are found as expected, with nothing except whitespace between them, it returns `t`; otherwise it returns `nil`.

This function cannot tell whether the comments it traverses are embedded within a string. If they look like comments, it treats them as comments.

To move forward over all comments and whitespace following point, use `(forward-comment (buffer-size))`. `(buffer-size)` is a good argument to use, because the number of comments in the buffer cannot exceed that many.

36.6.2 Finding the Parse State for a Position

For syntactic analysis, such as in indentation, often the useful thing is to compute the syntactic state corresponding to a given buffer position. This function does that conveniently.

`syntax-ppss` **&optional** *pos* [Function]

This function returns the parser state that the parser would reach at position *pos* starting from the beginning of the visible portion of the buffer. See the next section for a description of the parser state.

The return value is the same as if you call the low-level parsing function `parse-partial-sexp` to parse from the beginning of the visible portion of the buffer to *pos* (see Section 36.6.4 [Low-Level Parsing], page 1036). However, `syntax-ppss` uses caches to speed up the computation. Due to this optimization, the second value (previous complete subexpression) and sixth value (minimum parenthesis depth) in the returned parser state are not meaningful.

This function has a side effect: it adds a buffer-local entry to `before-change-functions` (see Section 33.34 [Change Hooks], page 964) for `syntax-ppss-flush-cache` (see below). This entry keeps the cache consistent as the buffer is modified. However, the cache might not be updated if `syntax-ppss` is called while `before-change-functions` is temporarily let-bound, or if the buffer is modified without running the hook, such as when using `inhibit-modification-hooks`. In those cases, it is necessary to call `syntax-ppss-flush-cache` explicitly.

`syntax-ppss-flush-cache` *beg* **&rest** *ignored-args* [Function]

This function flushes the cache used by `syntax-ppss`, starting at position *beg*. The remaining arguments, *ignored-args*, are ignored; this function accepts them so that it can be directly used on hooks such as `before-change-functions` (see Section 33.34 [Change Hooks], page 964).

36.6.3 Parser State

A *parser state* is a list of (currently) eleven elements describing the state of the syntactic parser, after it parses the text between a specified starting point and a specified end point in the buffer using `parse-partial-sexp` (see Section 36.6.4 [Low-Level Parsing], page 1036). Parsing functions such as `syntax-ppss` also return a parser state as the value. `parse-partial-sexp` can accept a parser state as an argument, for resuming parsing.

Here are the meanings of the elements of the parser state:

0. The depth in parentheses, counting from 0. **Warning:** this can be negative if there are more close parens than open parens between the parser's starting point and end point.

1. The character position of the start of the innermost parenthetical grouping containing the stopping point; `nil` if none.
2. The character position of the start of the last complete subexpression terminated; `nil` if none.
3. Non-`nil` if inside a string. More precisely, this is the character that will terminate the string, or `t` if a generic string delimiter character should terminate it.
4. `t` if inside a non-nestable comment (of any comment style; see Section 36.2.2 [Syntax Flags], page 1029); or the comment nesting level if inside a comment that can be nested.
5. `t` if the end point is just after a quote character.
6. The minimum parenthesis depth encountered during this scan.
7. What kind of comment is active: `nil` if not in a comment or in a comment of style 'a'; 1 for a comment of style 'b'; 2 for a comment of style 'c'; and `syntax-table` for a comment that should be ended by a generic comment delimiter character.
8. The string or comment start position. While inside a comment, this is the position where the comment began; while inside a string, this is the position where the string began. When outside of strings and comments, this element is `nil`.
9. The list of the positions of the currently open parentheses, starting with the outermost.
10. When the last buffer position scanned was the (potential) first character of a two character construct (comment delimiter or escaped/char-quoted character pair), the *syntax-code* (see Section 36.7 [Syntax Table Internals], page 1037) of that position. Otherwise `nil`.

Elements 1, 2, and 6 are ignored in a state which you pass as an argument to `parse-partial-sexp` to continue parsing. Elements 9 and 10 are mainly used internally by the parser code.

Some additional useful information is available from a parser state using these functions:

`syntax-ppss-toplevel-pos` *state* [Function]

This function extracts, from parser state *state*, the last position scanned in the parse which was at top level in grammatical structure. "At top level" means outside of any parentheses, comments, or strings.

The value is `nil` if *state* represents a parse which has arrived at a top level position.

`syntax-ppss-context` *state* [Function]

Return `string` if the end position of the scan returning *state* is in a string, and `comment` if it's in a comment. Otherwise return `nil`.

36.6.4 Low-Level Parsing

The most basic way to use the expression parser is to tell it to start at a given position with a certain state, and parse up to a specified end position.

`parse-partial-sexp` *start limit* **&optional** *target-depth stop-before state stop-comment* [Function]

This function parses a sexp in the current buffer starting at *start*, not scanning past *limit*. It stops at position *limit* or when certain criteria described below are met, and

sets point to the location where parsing stops. It returns a parser state describing the status of the parse at the point where it stops.

If the third argument *target-depth* is non-`nil`, parsing stops if the depth in parentheses becomes equal to *target-depth*. The depth starts at 0, or at whatever is given in *state*.

If the fourth argument *stop-before* is non-`nil`, parsing stops when it comes to any character that starts a sexp. If *stop-comment* is non-`nil`, parsing stops after the start of an unnested comment. If *stop-comment* is the symbol `syntax-table`, parsing stops after the start of an unnested comment or a string, or after the end of an unnested comment or a string, whichever comes first.

If *state* is `nil`, *start* is assumed to be at the top level of parenthesis structure, such as the beginning of a function definition. Alternatively, you might wish to resume parsing in the middle of the structure. To do this, you must provide a *state* argument that describes the initial status of parsing. The value returned by a previous call to `parse-partial-sexp` will do nicely.

36.6.5 Parameters to Control Parsing

`multibyte-syntax-as-symbol` [Variable]

If this variable is non-`nil`, `scan-sexps` treats all non-ASCII characters as symbol constituents regardless of what the syntax table says about them. (However, `syntax-table` text properties can still override the syntax.)

`parse-sexp-ignore-comments` [User Option]

If the value is non-`nil`, then comments are treated as whitespace by the functions in this section and by `forward-sexp`, `scan-lists` and `scan-sexps`.

The behavior of `parse-partial-sexp` is also affected by `parse-sexp-lookup-properties` (see Section 36.4 [Syntax Properties], page 1032).

`comment-end-can-be-escaped` [Variable]

If this buffer local variable is non-`nil`, a single character which usually terminates a comment doesn't do so when that character is escaped. This is used in C and C++ Modes, where line comments starting with '`//`' can be continued onto the next line by escaping the newline with '`\`'.

You can use `forward-comment` to move forward or backward over one comment or several comments.

36.7 Syntax Table Internals

Syntax tables are implemented as char-tables (see Section 6.6 [Char-Tables], page 119), but most Lisp programs don't work directly with their elements. Syntax tables do not store syntax data as syntax descriptors (see Section 36.2 [Syntax Descriptors], page 1026); they use an internal format, which is documented in this section. This internal format can also be assigned as syntax properties (see Section 36.4 [Syntax Properties], page 1032).

Each entry in a syntax table is a *raw syntax descriptor*: a cons cell of the form `(`*`syntax-code`*` . `*`matching-char`*`)`. *syntax-code* is an integer which encodes the syntax class and

syntax flags, according to the table below. *matching-char*, if non-`nil`, specifies a matching character (similar to the second character in a syntax descriptor).

Use `aref` (see Section 6.3 [Array Functions], page 116) to get the raw syntax descriptor of a character, e.g. `(aref (syntax-table) ch)`.

Here are the syntax codes corresponding to the various syntax classes:

Code	*Class*	*Code*	*Class*
0	whitespace	8	paired delimiter
1	punctuation	9	escape
2	word	10	character quote
3	symbol	11	comment-start
4	open parenthesis	12	comment-end
5	close parenthesis	13	inherit
6	expression prefix	14	generic comment
7	string quote	15	generic string

For example, in the standard syntax table, the entry for '`(`' is `(4 . 41)`. 41 is the character code for '`)`'.

Syntax flags are encoded in higher order bits, starting 16 bits from the least significant bit. This table gives the power of two which corresponds to each syntax flag.

Prefix	*Flag*	*Prefix*	*Flag*
'`1`'	`(ash 1 16)`	'`p`'	`(ash 1 20)`
'`2`'	`(ash 1 17)`	'`b`'	`(ash 1 21)`
'`3`'	`(ash 1 18)`	'`n`'	`(ash 1 22)`
'`4`'	`(ash 1 19)`	'`c`'	`(ash 1 23)`

`string-to-syntax` *desc* [Function]
Given a syntax descriptor *desc* (a string), this function returns the corresponding raw syntax descriptor.

`syntax-class-to-char` *syntax* [Function]
Given a raw syntax descriptor *syntax* (an integer), this function returns the corresponding syntax descriptor (a character).

`syntax-after` *pos* [Function]
This function returns the raw syntax descriptor for the character in the buffer after position *pos*, taking account of syntax properties as well as the syntax table. If *pos* is outside the buffer's accessible portion (see Section 31.4 [Narrowing], page 867), the return value is `nil`.

`syntax-class` *syntax* [Function]
This function returns the syntax code for the raw syntax descriptor *syntax*. More precisely, it takes the raw syntax descriptor's *syntax-code* component, masks off the high 16 bits which record the syntax flags, and returns the resulting integer.

If *syntax* is `nil`, the return value is `nil`. This is so that the expression

```
(syntax-class (syntax-after pos))
```

evaluates to `nil` if `pos` is outside the buffer's accessible portion, without throwing errors or returning an incorrect code.

36.8 Categories

Categories provide an alternate way of classifying characters syntactically. You can define several categories as needed, then independently assign each character to one or more categories. Unlike syntax classes, categories are not mutually exclusive; it is normal for one character to belong to several categories.

Each buffer has a *category table* which records which categories are defined and also which characters belong to each category. Each category table defines its own categories, but normally these are initialized by copying from the standard categories table, so that the standard categories are available in all modes.

Each category has a name, which is an ASCII printing character in the range ' ' to '~'. You specify the name of a category when you define it with `define-category`.

The category table is actually a char-table (see Section 6.6 [Char-Tables], page 119). The element of the category table at index *c* is a *category set*—a bool-vector—that indicates which categories character *c* belongs to. In this category set, if the element at index *cat* is `t`, that means category *cat* is a member of the set, and that character *c* belongs to category *cat*.

For the next three functions, the optional argument *table* defaults to the current buffer's category table.

`define-category` *char docstring* **&optional** *table* [Function]

This function defines a new category, with name *char* and documentation *docstring*, for the category table *table*.

Here's an example of defining a new category for characters that have strong right-to-left directionality (see Section 41.27 [Bidirectional Display], page 1255) and using it in a special category table. To obtain the information about the directionality of characters, the example code uses the '`bidi-class`' Unicode property (see Section 34.6 [Character Properties], page 971).

```
(defvar special-category-table-for-bidi
  ;;     Make an empty category-table.
  (let ((category-table (make-category-table))
        ;; Create a char-table which gives the 'bidi-class' Unicode
        ;; property for each character.
        (uniprop-table
         (unicode-property-table-internal 'bidi-class)))
    (define-category ?R "Characters of bidi-class R, AL, or RLO"
                     category-table)
    ;; Modify the category entry of each character whose
    ;; 'bidi-class' Unicode property is R, AL, or RLO --
    ;; these have a right-to-left directionality.
    (map-char-table
     (lambda (key val)
       (if (memq val '(R AL RLO))
           (modify-category-entry key ?R category-table)))
     uniprop-table)
    category-table))
```

`category-docstring` *category* **&optional** *table* [Function]
This function returns the documentation string of category *category* in category table *table*.

```
(category-docstring ?a)
     ⇒ "ASCII"
(category-docstring ?l)
     ⇒ "Latin"
```

`get-unused-category` **&optional** *table* [Function]
This function returns a category name (a character) which is not currently defined in *table*. If all possible categories are in use in *table*, it returns `nil`.

`category-table` [Function]
This function returns the current buffer's category table.

`category-table-p` *object* [Function]
This function returns `t` if *object* is a category table, otherwise `nil`.

`standard-category-table` [Function]
This function returns the standard category table.

`copy-category-table` **&optional** *table* [Function]
This function constructs a copy of *table* and returns it. If *table* is not supplied (or is `nil`), it returns a copy of the standard category table. Otherwise, an error is signaled if *table* is not a category table.

`set-category-table` *table* [Function]
This function makes *table* the category table for the current buffer. It returns *table*.

`make-category-table` [Function]
This creates and returns an empty category table. In an empty category table, no categories have been allocated, and no characters belong to any categories.

`make-category-set` *categories* [Function]
This function returns a new category set—a bool-vector—whose initial contents are the categories listed in the string *categories*. The elements of *categories* should be category names; the new category set has `t` for each of those categories, and `nil` for all other categories.

```
(make-category-set "al")
     ⇒ #&128"\0\0\0\0\0\0\0\0\0\0\0\0\2\20\0\0"
```

`char-category-set` *char* [Function]
This function returns the category set for character *char* in the current buffer's category table. This is the bool-vector which records which categories the character *char* belongs to. The function `char-category-set` does not allocate storage, because it returns the same bool-vector that exists in the category table.

```
(char-category-set ?a)
     ⇒ #&128"\0\0\0\0\0\0\0\0\0\0\0\0\2\20\0\0"
```

`category-set-mnemonics` *category-set* [Function]

This function converts the category set *category-set* into a string containing the characters that designate the categories that are members of the set.

```
(category-set-mnemonics (char-category-set ?a))
     ⇒ "al"
```

`modify-category-entry` *char category* **&optional** *table reset* [Function]

This function modifies the category set of *char* in category table *table* (which defaults to the current buffer's category table). *char* can be a character, or a cons cell of the form `(`*min* `.` *max*`)`; in the latter case, the function modifies the category sets of all characters in the range between *min* and *max*, inclusive.

Normally, it modifies a category set by adding *category* to it. But if *reset* is non-`nil`, then it deletes *category* instead.

`describe-categories` **&optional** *buffer-or-name* [Command]

This function describes the category specifications in the current category table. It inserts the descriptions in a buffer, and then displays that buffer. If *buffer-or-name* is non-`nil`, it describes the category table of that buffer instead.

37 Parsing Program Source

Emacs provides various ways to parse program source text and produce a *syntax tree*. In a syntax tree, text is no longer considered a one-dimensional stream of characters, but a structured tree of nodes, where each node represents a piece of text. Thus, a syntax tree can enable interesting features like precise fontification, indentation, navigation, structured editing, etc.

Emacs has a simple facility for parsing balanced expressions (see Section 36.6 [Parsing Expressions], page 1033). There is also the SMIE library for generic navigation and indentation (see Section 24.7.1 [SMIE], page 583).

In addition to those, Emacs also provides integration with the tree-sitter library (`https://tree-sitter.github.io/tree-sitter`) if support for it was compiled in. The tree-sitter library implements an incremental parser and has support for a wide range of programming languages.

`treesit-available-p` [Function]

This function returns non-`nil` if tree-sitter features are available for the current Emacs session.

To be able to parse the program source using the tree-sitter library and access the syntax tree of the program, a Lisp program needs to load a language grammar library, and create a parser for that language and the current buffer. After that, the Lisp program can query the parser about specific nodes of the syntax tree. Then, it can access various kinds of information about each node, and search for nodes using a powerful pattern-matching syntax. This chapter explains how to do all this, and also how a Lisp program can work with source files that mix multiple programming languages.

37.1 Tree-sitter Language Grammar

Loading a language grammar

Tree-sitter relies on language grammar to parse text in that language. In Emacs, a language grammar is represented by a symbol. For example, the C language grammar is represented as the symbol `c`, and `c` can be passed to tree-sitter functions as the *language* argument.

Tree-sitter language grammars are distributed as dynamic libraries. In order to use a language grammar in Emacs, you need to make sure that the dynamic library is installed on the system. Emacs looks for language grammars in several places, in the following order:

- first, in the list of directories specified by the variable `treesit-extra-load-path`;
- then, in the `tree-sitter` subdirectory of the directory specified by `user-emacs-directory` (see Section 42.1.2 [Init File], page 1263);
- and finally, in the system's default locations for dynamic libraries.

In each of these directories, Emacs looks for a file with file-name extensions specified by the variable `dynamic-library-suffixes`.

If Emacs cannot find the library or has problems loading it, Emacs signals the `treesit-load-language-error` error. The data of that signal could be one of the following:

`(not-found` *`error-msg ...`*`)`
: This means that Emacs could not find the language grammar library.

`(symbol-error` *`error-msg`*`)`
: This means that Emacs could not find in the library the expected function that every language grammar library should export.

`(version-mismatch` *`error-msg`*`)`
: This means that the version of the language grammar library is incompatible with that of the tree-sitter library.

In all of these cases, *error-msg* might provide additional details about the failure.

`treesit-language-available-p` *language* **&optional** *detail* [Function]
: This function returns non-`nil` if the language grammar for *language* exists and can be loaded.

  If *detail* is non-`nil`, return `(t . nil)` when *language* is available, and `(nil .` *`data`*`)` when it's unavailable. *data* is the signal data of `treesit-load-language-error`.

By convention, the file name of the dynamic library for *language* is `libtree-sitter-`*`language.ext`*, where *ext* is the system-specific extension for dynamic libraries. Also by convention, the function provided by that library is named `tree_sitter_`*`language`*. If a language grammar library doesn't follow this convention, you should add an entry

```
(language library-base-name function-name)
```

to the list in the variable `treesit-load-name-override-list`, where *library-base-name* is the basename of the dynamic library's file name (usually, `libtree-sitter-`*`language`*), and *function-name* is the function provided by the library (usually, `tree_sitter_`*`language`*). For example,

```
(cool-lang "libtree-sitter-coool" "tree_sitter_cooool")
```

for a language that considers itself too "cool" to abide by conventions.

`treesit-library-abi-version` **&optional** *min-compatible* [Function]
: This function returns the version of the language grammar Application Binary Interface (ABI) supported by the tree-sitter library. By default, it returns the latest ABI version supported by the library, but if *min-compatible* is non-`nil`, it returns the oldest ABI version which the library still can support. Language grammar libraries must be built for ABI versions between the oldest and the latest versions supported by the tree-sitter library, otherwise the library will be unable to load them.

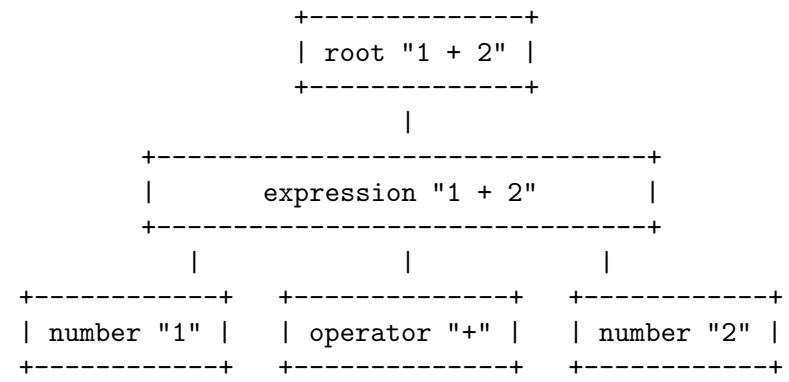
`treesit-language-abi-version` *language* [Function]
: This function returns the ABI version of the language grammar library loaded by Emacs for *language*. If *language* is unavailable, this function returns `nil`.

Concrete syntax tree

A syntax tree is what a parser generates. In a syntax tree, each node represents a piece of text, and is connected to each other by a parent-child relationship. For example, if the source text is

```
1 + 2
```

its syntax tree could be



We can also represent it as an s-expression:

```
(root (expression (number) (operator) (number)))
```

Node types

Names like `root`, `expression`, `number`, and `operator` specify the *type* of the nodes. However, not all nodes in a syntax tree have a type. Nodes that don't have a type are known as *anonymous nodes*, and nodes with a type are *named nodes*. Anonymous nodes are tokens with fixed spellings, including punctuation characters like bracket '`]`', and keywords like `return`.

Field names

To make the syntax tree easier to analyze, many language grammar assign *field names* to child nodes. For example, a `function_definition` node could have a `declarator` and a `body`:

```
(function_definition
 declarator: (declaration)
 body: (compound_statement))
```

Exploring the syntax tree

To aid in understanding the syntax of a language and in debugging Lisp programs that use the syntax tree, Emacs provides an "explore" mode, which displays the syntax tree of the source in the current buffer in real time. Emacs also comes with an "inspect mode", which displays information of the nodes at point in the mode-line.

`treesit-explore-mode` [Command]

This mode pops up a window displaying the syntax tree of the source in the current buffer. Selecting text in the source buffer highlights the corresponding nodes in the syntax tree display. Clicking on nodes in the syntax tree highlights the corresponding text in the source buffer.

`treesit-inspect-mode` [Command]

This minor mode displays on the mode-line the node that *starts* at point. For example, the mode-line can display

parent field: (*node* (*child* (...)))

where *node*, *child*, etc., are nodes which begin at point. *parent* is the parent of *node*. *node* is displayed in a bold typeface. *field-name*s are field names of *node* and of *child*, etc.

If no node starts at point, i.e., point is in the middle of a node, then the mode line displays the earliest node that spans point, and its immediate parent.

This minor mode doesn't create parsers on its own. It uses the first parser in `(treesit-parser-list)` (see Section 37.2 [Using Parser], page 1046).

Reading the grammar definition

Authors of language grammars define the *grammar* of a programming language, which determines how a parser constructs a concrete syntax tree out of the program text. In order to use the syntax tree effectively, you need to consult the *grammar file*.

The grammar file is usually `grammar.js` in a language grammar's project repository. The link to a language grammar's home page can be found on tree-sitter's homepage (`https://tree-sitter.github.io/tree-sitter`).

The grammar definition is written in JavaScript. For example, the rule matching a `function_definition` node may look like

```
function_definition: $ => seq(
  $.declaration_specifiers,
  field('declarator', $.declaration),
  field('body', $.compound_statement)
)
```

The rules are represented by functions that take a single argument *$*, representing the whole grammar. The function itself is constructed by other functions: the `seq` function puts together a sequence of children; the `field` function annotates a child with a field name. If we write the above definition in the so-called *Backus-Naur Form* (BNF) syntax, it would look like

```
function_definition :=
  <declaration_specifiers> <declaration> <compound_statement>
```

and the node returned by the parser would look like

```
(function_definition
  (declaration_specifier)
  declarator: (declaration)
  body: (compound_statement))
```

Below is a list of functions that one can see in a grammar definition. Each function takes other rules as arguments and returns a new rule.

`seq(`*`rule1`*`, `*`rule2`*`, ...)`
: matches each rule one after another.

`choice(`*`rule1`*`, `*`rule2`*`, ...)`
: matches one of the rules in its arguments.

`repeat(`*`rule`*`)`
: matches *rule zero or more* times. This is like the '`*`' operator in regular expressions.

`repeat1(rule)`
: matches *rule one or more* times. This is like the '`+`' operator in regular expressions.

`optional(rule)`
: matches *rule zero or one* times. This is like the '`?`' operator in regular expressions.

`field(name, rule)`
: assigns field name *name* to the child node matched by *rule*.

`alias(rule, alias)`
: makes nodes matched by *rule* appear as *alias* in the syntax tree generated by the parser. For example,

```
alias(preprocessor_call_exp, call_expression)
```

: makes any node matched by `preprocessor_call_exp` appear as `call_expression`.

Below are grammar functions of lesser importance for reading a language grammar.

`token(rule)`
: marks *rule* to produce a single leaf node. That is, instead of generating a parent node with individual child nodes under it, everything is combined into a single leaf node. See Section 37.3 [Retrieving Nodes], page 1048.

`token.immediate(rule)`
: Normally, grammar rules ignore preceding whitespace; this changes *rule* to match only when there is no preceding whitespace.

`prec(n, rule)`
: gives *rule* the level-*n* precedence.

`prec.left([n,] rule)`
: marks *rule* as left-associative, optionally with level *n*.

`prec.right([n,] rule)`
: marks *rule* as right-associative, optionally with level *n*.

`prec.dynamic(n, rule)`
: this is like `prec`, but the precedence is applied at runtime instead.

The documentation of the tree-sitter project has more about writing a grammar (`https://tree-sitter.github.io/tree-sitter/creating-parsers`). Read especially "The Grammar DSL" section.

37.2 Using Tree-sitter Parser

This section describes how to create and configure a tree-sitter parser. In Emacs, each tree-sitter parser is associated with a buffer. As the user edits the buffer, the associated parser and syntax tree are automatically kept up-to-date.

`treesit-max-buffer-size` [Variable]
: This variable contains the maximum size of buffers in which tree-sitter can be activated. Major modes should check this value when deciding whether to enable tree-sitter features.

`treesit-parser-create` *language* **&optional** *buffer no-reuse* [Function]
Create a parser for the specified *buffer* and *language* (see Section 37.1 [Language Grammar], page 1042). If *buffer* is omitted or `nil`, it stands for the current buffer.

By default, this function reuses a parser if one already exists for *language* in *buffer*, but if *no-reuse* is non-`nil`, this function always creates a new parser.

If that buffer is an indirect buffer, its base buffer is used instead. That is, indirect buffers use their base buffer's parsers. If the base buffer is narrowed, an indirect buffer might not be able to retrieve information of the portion of the buffer text that is invisible in the base buffer. Lisp programs should widen as necessary should they want to use a parser in an indirect buffer.

Given a parser, we can query information about it.

`treesit-parser-buffer` *parser* [Function]
This function returns the buffer associated with *parser*.

`treesit-parser-language` *parser* [Function]
This function returns the language used by *parser*.

`treesit-parser-p` *object* [Function]
This function checks if *object* is a tree-sitter parser, and returns non-`nil` if it is, and `nil` otherwise.

There is no need to explicitly parse a buffer, because parsing is done automatically and lazily. A parser only parses when a Lisp program queries for a node in its syntax tree. Therefore, when a parser is first created, it doesn't parse the buffer; it waits until the Lisp program queries for a node for the first time. Similarly, when some change is made in the buffer, a parser doesn't re-parse immediately.

When a parser does parse, it checks for the size of the buffer. Tree-sitter can only handle buffers no larger than about 4GB. If the size exceeds that, Emacs signals the `treesit-buffer-too-large` error with signal data being the buffer size.

Once a parser is created, Emacs automatically adds it to the internal parser list. Every time a change is made to the buffer, Emacs updates parsers in this list so they can update their syntax tree incrementally.

`treesit-parser-list` **&optional** *buffer* [Function]
This function returns the parser list of *buffer*. If *buffer* is `nil` or omitted, it defaults to the current buffer. If that buffer is an indirect buffer, its base buffer is used instead. That is, indirect buffers use their base buffer's parsers.

`treesit-parser-delete` *parser* [Function]
This function deletes *parser*.

Normally, a parser "sees" the whole buffer, but when the buffer is narrowed (see Section 31.4 [Narrowing], page 867), the parser will only see the accessible portion of the buffer. As far as the parser can tell, the hidden region was deleted. When the buffer is later widened, the parser thinks text is inserted at the beginning and at the end. Although

parsers respect narrowing, modes should not use narrowing as a means to handle a multi-language buffer; instead, set the ranges in which the parser should operate. See Section 37.6 [Multiple Languages], page 1060.

Because a parser parses lazily, when the user or a Lisp program narrows the buffer, the parser is not affected immediately; as long as the mode doesn't query for a node while the buffer is narrowed, the parser is oblivious of the narrowing.

Besides creating a parser for a buffer, a Lisp program can also parse a string. Unlike a buffer, parsing a string is a one-off operation, and there is no way to update the result.

`treesit-parse-string` *string language* [Function]
: This function parses *string* using *language*, and returns the root node of the generated syntax tree.

Be notified by changes to the parse tree

A Lisp program might want to be notified of text affected by incremental parsing. For example, inserting a comment-closing token converts text before that token into a comment. Even though the text is not directly edited, it is deemed to be "changed" nevertheless.

Emacs lets a Lisp program register callback functions (a.k.a. *notifiers*) for these kinds of changes. A notifier function takes two arguments: *ranges* and *parser*. *ranges* is a list of cons cells of the form `(start . end)`, where *start* and *end* mark the start and the end positions of a range. *parser* is the parser issuing the notification.

Every time a parser reparses a buffer, it compares the old and new parse-tree, computes the ranges in which nodes have changed, and passes the ranges to notifier functions. Note that the initial parse is also considered a "change", so notifier functions are called on the initial parse, with range being the whole buffer.

`treesit-parser-add-notifier` *parser function* [Function]
: This function adds *function* to *parser*'s list of after-change notifier functions. *function* must be a function symbol, not a lambda function (see Section 13.7 [Anonymous Functions], page 243).

`treesit-parser-remove-notifier` *parser function* [Function]
: This function removes *function* from the list of *parser*'s after-change notifier functions. *function* must be a function symbol, rather than a lambda function.

`treesit-parser-notifiers` *parser* [Function]
: This function returns the list of *parser*'s notifier functions.

37.3 Retrieving Nodes

Here are some terms and conventions we use when documenting tree-sitter functions.

A node in a syntax tree spans some portion of the program text in the buffer. We say that a node is "smaller" or "larger" than another if it spans, respectively, a smaller or larger portion of buffer text than the other node. Since nodes that are deeper ("lower") in the tree are children of the nodes that are "higher" in the tree, it follows that a lower node will always be smaller than a node that is higher in the node hierarchy. A node that is higher

up in the syntax tree contains one or more smaller nodes as its children, and therefore spans a larger portion of buffer text.

When a function cannot find a node, it returns `nil`. For convenience, all functions that take a node as argument and return a node, also accept the node argument of `nil` and in that case just return `nil`.

Nodes are not automatically updated when the associated buffer is modified, and there is no way to update a node once it is retrieved. Using an outdated node signals the `treesit-node-outdated` error.

Retrieving nodes from syntax tree

`treesit-node-at` *pos* **&optional** *parser-or-lang named* [Function]

This function returns a *leaf* node at buffer position *pos*. A leaf node is a node that doesn't have any child nodes.

This function tries to return a node whose span covers *pos*: the node's beginning position is less than or equal to *pos*, and the node's end position is greater than or equal to *pos*.

If no leaf node's span covers *pos* (e.g., *pos* is in the whitespace between two leaf nodes), this function returns the first leaf node after *pos*.

Finally, if there is no leaf node after *pos*, return the first leaf node before *pos*.

If *parser-or-lang* is a parser object, this function uses that parser; if *parser-or-lang* is a language, this function uses the first parser for that language in the current buffer, or creates one if none exists; if *parser-or-lang* is `nil`, this function tries to guess the language at *pos* by calling `treesit-language-at` (see Section 37.6 [Multiple Languages], page 1060).

If this function cannot find a suitable node to return, it returns `nil`.

If *named* is non-`nil`, this function looks only for named nodes (see [tree-sitter named node], page 1044).

Example:

```
;; Find the node at point in a C parser's syntax tree.
(treesit-node-at (point) 'c)
  ⇒ #<treesit-node (primitive_type) in 23-27>
```

`treesit-node-on` *beg end* **&optional** *parser-or-lang named* [Function]

This function returns the *smallest* node that covers the region of buffer text between *beg* and *end*. In other words, the start of the node is before or at *beg*, and the end of the node is at or after *end*.

Beware: calling this function on an empty line that is not inside any top-level construct (function definition, etc.) most probably will give you the root node, because the root node is the smallest node that covers that empty line. Most of the time, you want to use `treesit-node-at` instead.

If *parser-or-lang* is a parser object, this function uses that parser; if *parser-or-lang* is a language, this function uses the first parser for that language in the current buffer, or creates one if none exists; if *parser-or-lang* is `nil`, this function tries to guess the language at *beg* by calling `treesit-language-at`.

If *named* is non-`nil`, this function looks for a named node only (see [tree-sitter named node], page 1044).

`treesit-parser-root-node` *parser* [Function]
This function returns the root node of the syntax tree generated by *parser*.

`treesit-buffer-root-node` **&optional** *language* [Function]
This function finds the first parser for *language* in the current buffer, or creates one if none exists, and returns the root node generated by that parser. If *language* is omitted, it uses the first parser in the parser list. If it cannot find an appropriate parser, it returns `nil`.

Given a node, a Lisp program can retrieve other nodes starting from it, or query for information about this node.

Retrieving nodes from other nodes

By kinship

`treesit-node-parent` *node* [Function]
This function returns the immediate parent of *node*.

If *node* is more than 1000 levels deep in a parse tree, the return value is undefined. Currently it returns `nil`, but that could change in the future.

`treesit-node-child` *node n* **&optional** *named* [Function]
This function returns the *n*'th child of *node*. If *named* is non-`nil`, it counts only named nodes (see [tree-sitter named node], page 1044).

For example, in a node that represents a string `"text"`, there are three children nodes: the opening quote `"`, the string text `text`, and the closing quote `"`. Among these nodes, the first child is the opening quote `"`, and the first named child is the string text.

This function returns `nil` if there is no *n*'th child. *n* could be negative, e.g., -1 represents the last child.

`treesit-node-children` *node* **&optional** *named* [Function]
This function returns all of *node*'s children as a list. If *named* is non-`nil`, it retrieves only named nodes.

`treesit-node-next-sibling` *node* **&optional** *named* [Function]
This function finds the next sibling of *node*. If *named* is non-`nil`, it finds the next named sibling.

`treesit-node-prev-sibling` *node* **&optional** *named* [Function]
This function finds the previous sibling of *node*. If *named* is non-`nil`, it finds the previous named sibling.

By field name

To make the syntax tree easier to analyze, many language grammars assign *field names* to child nodes (see [tree-sitter node field name], page 1044). For example, a `function_definition` node could have a `declarator` node and a `body` node.

`treesit-node-child-by-field-name` *node field-name* [Function]

This function finds the child of *node* whose field name is *field-name*, a string.

```
;; Get the child that has "body" as its field name.
(treesit-node-child-by-field-name node "body")
  ⇒ #<treesit-node (compound_statement) in 45-89>
```

By position

`treesit-node-first-child-for-pos` *node pos* **&optional** *named* [Function]

This function finds the first child of *node* that extends beyond buffer position *pos*. "Extends beyond" means the end of the child node is greater or equal to *pos*. This function only looks for immediate children of *node*, and doesn't look in its grandchildren. If *named* is non-`nil`, it looks for the first named child (see [tree-sitter named node], page 1044).

`treesit-node-descendant-for-range` *node beg end* **&optional** *named* [Function]

This function finds the *smallest* descendant node of *node* that spans the region of text between positions *beg* and *end*. It is similar to `treesit-node-at`. If *named* is non-`nil`, it looks for the smallest named child.

Searching for node

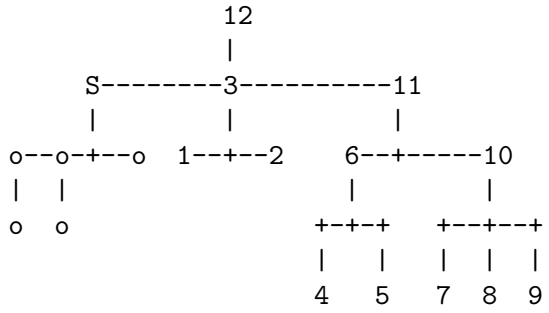
`treesit-search-subtree` *node predicate* **&optional** *backward all depth* [Function]

This function traverses the subtree of *node* (including *node* itself), looking for a node for which *predicate* returns non-`nil`. *predicate* is a regexp that is matched against each node's type, or a predicate function that takes a node and returns non-`nil` if the node matches. The function returns the first node that matches, or `nil` if none does.

By default, this function only traverses named nodes, but if *all* is non-`nil`, it traverses all the nodes. If *backward* is non-`nil`, it traverses backwards (i.e., it visits the last child first when traversing down the tree). If *depth* is non-`nil`, it must be a number that limits the tree traversal to that many levels down the tree. If *depth* is `nil`, it defaults to 1000.

`treesit-search-forward` *start predicate* **&optional** *backward all* [Function]

Like `treesit-search-subtree`, this function also traverses the parse tree and matches each node with *predicate* (except for *start*), where *predicate* can be a regexp or a function. For a tree like the one below where *start* is marked '`S`', this function traverses as numbered from 1 to 12:



Note that this function doesn't traverse the subtree of *start*, and it always traverses leaf nodes first, before moving upwards.

Like `treesit-search-subtree`, this function only searches for named nodes by default, but if *all* is non-`nil`, it searches for all nodes. If *backward* is non-`nil`, it searches backwards.

While `treesit-search-subtree` traverses the subtree of a node, this function starts with node *start* and traverses every node that comes after it in the buffer position order, i.e., nodes with start positions greater than the end position of *start*.

In the tree shown above, `treesit-search-subtree` traverses node '`S`' (*start*) and nodes marked with `o`, whereas this function traverses the nodes marked with numbers. This function is useful for answering questions like "what is the first node after *start* in the buffer that satisfies some condition?"

`treesit-search-forward-goto` *node predicate* **&optional** *start backward all* [Function]

This function moves point to the start or end of the next node after *node* in the buffer that matches *predicate*. If *start* is non-`nil`, stop at the beginning rather than the end of a node.

This function guarantees that the matched node it returns makes progress in terms of buffer position: the start/end position of the returned node is always greater than that of *node*.

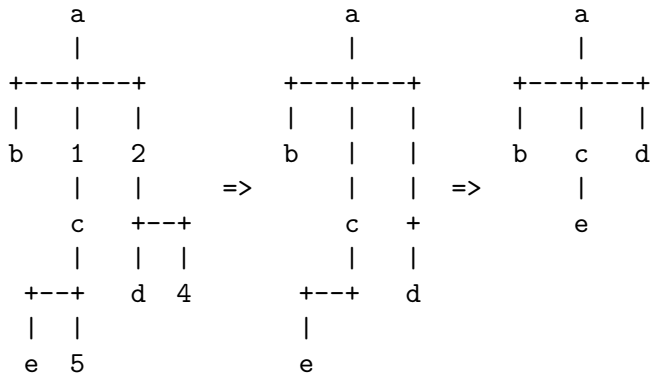
Arguments *predicate*, *backward*, and *all* are the same as in `treesit-search-forward`.

`treesit-induce-sparse-tree` *root predicate* **&optional** *process-fn depth* [Function]

This function creates a sparse tree from *root*'s subtree.

It takes the subtree under *root*, and combs it so only the nodes that match *predicate* are left. Like previous functions, the *predicate* can be a regexp string that matches against each node's type, or a function that takes a node and returns non-`nil` if it matches.

For example, given the subtree on the left that consists of both numbers and letters, if *predicate* is "letter only", the returned tree is the one on the right.



If *process-fn* is non-`nil`, instead of returning the matched nodes, this function passes each node to *process-fn* and uses the returned value instead. If non-`nil`, *depth* limits the number of levels to go down from *root*. If *depth* is `nil`, it defaults to 1000.

Each node in the returned tree looks like (*tree-sitter-node* . (*child* ...)). The *tree-sitter-node* of the root of this tree will be `nil` if *root* doesn't match *predicate*. If no node matches *predicate*, the function returns `nil`.

More convenience functions

`treesit-filter-child` *node predicate* **&optional** *named* [Function]

This function finds immediate children of *node* that satisfy *predicate*.

The *predicate* function takes a node as argument and should return non-`nil` to indicate that the node should be kept. If *named* is non-`nil`, this function only examines named nodes.

`treesit-parent-until` *node predicate* **&optional** *include-node* [Function]

This function repeatedly finds the parents of *node*, and returns the parent that satisfies *predicate*, a function that takes a node as argument and returns a boolean that indicates a match. If no parent satisfies *predicate*, this function returns `nil`.

Normally this function only looks at the parents of *node* but not *node* itself. But if *include-node* is non-`nil`, this function returns *node* if *node* satisfies *predicate*.

`treesit-parent-while` *node predicate* [Function]

This function goes up the tree starting from *node*, and keeps doing so as long as the nodes satisfy *predicate*, a function that takes a node as argument. That is, this function returns the highest parent of *node* that still satisfies *predicate*. Note that if *node* satisfies *predicate* but its immediate parent doesn't, *node* itself is returned.

`treesit-node-top-level` *node* **&optional** *predicate include-node* [Function]

This function returns the highest parent of *node* that has the same type as *node*. If no such parent exists, it returns `nil`. Therefore this function is also useful for testing whether *node* is top-level.

If *predicate* is `nil`, this function uses *node*'s type to find the parent. If *predicate* is non-`nil`, this function searches the parent that satisfies *predicate*. If *include-node* is non-`nil`, this function returns *node* if *node* satisfies *predicate*.

37.4 Accessing Node Information

Basic information of Node

Every node is associated with a parser, and that parser is associated with a buffer. The following functions retrieve them.

`treesit-node-parser` *node* [Function]
: This function returns *node*'s associated parser.

`treesit-node-buffer` *node* [Function]
: This function returns *node*'s parser's associated buffer.

`treesit-node-language` *node* [Function]
: This function returns *node*'s parser's associated language.

Each node represents a portion of text in the buffer. Functions below find relevant information about that text.

`treesit-node-start` *node* [Function]
: Return the start position of *node*.

`treesit-node-end` *node* [Function]
: Return the end position of *node*.

`treesit-node-text` *node* **&optional** *object* [Function]
: Return the buffer text that *node* represents, as a string. (If *node* is retrieved from parsing a string, it will be the text from that string.)

Here are some predicates on tree-sitter nodes:

`treesit-node-p` *object* [Function]
: Checks if *object* is a tree-sitter syntax node.

`treesit-node-eq` *node1 node2* [Function]
: Checks if *node1* and *node2* refer to the same node in a tree-sitter syntax tree. This function uses the same equivalence metric as `equal`. You can also compare nodes using `equal` (see Section 2.8 [Equality Predicates], page 34).

Property information

In general, nodes in a concrete syntax tree fall into two categories: *named nodes* and *anonymous nodes*. Whether a node is named or anonymous is determined by the language grammar (see [tree-sitter named node], page 1044).

Apart from being named or anonymous, a node can have other properties. A node can be "missing": such nodes are inserted by the parser in order to recover from certain kinds of syntax errors, i.e., something should probably be there according to the grammar, but is not there. This can happen during editing of the program source, when the source is not yet in its final form.

A node can be "extra": such nodes represent things like comments, which can appear anywhere in the text.

A node can be "outdated", if its parser has reparsed at least once after the node was created.

A node "has error" if the text it spans contains a syntax error. It can be that the node itself has an error, or one of its descendants has an error.

A node is considered *live* if its parser is not deleted, and the buffer to which it belongs is a live buffer (see Section 28.10 [Killing Buffers], page 687).

`treesit-node-check` *node property* [Function]
: This function returns non-`nil` if *node* has the specified *property*. *property* can be `named`, `missing`, `extra`, `outdated`, `has-error`, or `live`.

`treesit-node-type` *node* [Function]
: Named nodes have "types" (see [tree-sitter node type], page 1044). For example, a named node can be a `string_literal` node, where `string_literal` is its type. The type of an anonymous node is just the text that the node represents; e.g., the type of a '`,`' node is just '`,`'.

  This function returns *node*'s type as a string.

Information as a child or parent

`treesit-node-index` *node* **&optional** *named* [Function]
: This function returns the index of *node* as a child node of its parent. If *named* is non-`nil`, it only counts named nodes (see [tree-sitter named node], page 1044).

`treesit-node-field-name` *node* [Function]
: A child of a parent node could have a field name (see [tree-sitter node field name], page 1044). This function returns the field name of *node* as a child of its parent.

`treesit-node-field-name-for-child` *node n* [Function]
: This function returns the field name of the *n*'th child of *node*. It returns `nil` if there is no *n*'th child, or the *n*'th child doesn't have a field name.

  Note that *n* counts both named and anonymous children, and *n* can be negative, e.g., -1 represents the last child.

`treesit-node-child-count` *node* **&optional** *named* [Function]
: This function returns the number of children of *node*. If *named* is non-`nil`, it only counts named children (see [tree-sitter named node], page 1044).

37.5 Pattern Matching Tree-sitter Nodes

Tree-sitter lets Lisp programs match patterns using a small declarative language. This pattern matching consists of two steps: first tree-sitter matches a *pattern* against nodes in the syntax tree, then it *captures* specific nodes that matched the pattern and returns the captured nodes.

We describe first how to write the most basic query pattern and how to capture nodes in a pattern, then the pattern-matching function, and finally the more advanced pattern syntax.

Basic query syntax

A *query* consists of multiple *patterns*. Each pattern is an s-expression that matches a certain node in the syntax node. A pattern has the form `(type (child...))`.

For example, a pattern that matches a `binary_expression` node that contains `number_literal` child nodes would look like

```
(binary_expression (number_literal))
```

To *capture* a node using the query pattern above, append `@capture-name` after the node pattern you want to capture. For example,

```
(binary_expression (number_literal) @number-in-exp)
```

captures `number_literal` nodes that are inside a `binary_expression` node with the capture name `number-in-exp`.

We can capture the `binary_expression` node as well, with, for example, the capture name `biexp`:

```
(binary_expression
 (number_literal) @number-in-exp) @biexp
```

Query function

Now we can introduce the *query functions*.

`treesit-query-capture` *node query* **&optional** *beg end node-only* [Function]

This function matches patterns in *query* within *node*. The argument *query* can be either an s-expression, a string, or a compiled query object. For now, we focus on the s-expression syntax; string syntax and compiled queries are described at the end of the section.

The argument *node* can also be a parser or a language symbol. A parser means use its root node, a language symbol means find or create a parser for that language in the current buffer, and use the root node.

The function returns all the captured nodes in an alist with elements of the form `(capture_name . node)`. If *node-only* is non-`nil`, it returns the list of *node*s instead. By default the entire text of *node* is searched, but if *beg* and *end* are both non-`nil`, they specify the region of buffer text where this function should match nodes. Any matching node whose span overlaps with the region between *beg* and *end* is captured; it doesn't have to be completely contained in the region.

This function raises the `treesit-query-error` error if *query* is malformed. The signal data contains a description of the specific error. You can use `treesit-query-validate` to validate and debug the query.

For example, suppose *node*'s text is `1 + 2`, and *query* is

```
(setq query
      '((binary_expression
         (number_literal) @number-in-exp) @biexp)
```

Matching that query would return

```
(treesit-query-capture node query)
    ⇒ ((biexp . <node for "1 + 2">)
       (number-in-exp . <node for "1">)
       (number-in-exp . <node for "2">))
```

As mentioned earlier, *query* could contain multiple patterns. For example, it could have two top-level patterns:

```
(setq query
      '((binary_expression) @biexp
        (number_literal) @number @biexp)
```

`treesit-query-string` *string query language* [Function]
: This function parses *string* as *language*, matches its root node with *query*, and returns the result.

More query syntax

Besides node type and capture name, tree-sitter's pattern syntax can express anonymous node, field name, wildcard, quantification, grouping, alternation, anchor, and predicate.

Anonymous node

An anonymous node is written verbatim, surrounded by quotes. A pattern matching (and capturing) keyword `return` would be

```
"return" @keyword
```

Wild card

In a pattern, '`(_)`' matches any named node, and '`_`' matches any named or anonymous node. For example, to capture any named child of a `binary_expression` node, the pattern would be

```
(binary_expression (_) @in-biexp)
```

Field name

It is possible to capture child nodes that have specific field names. In the pattern below, `declarator` and `body` are field names, indicated by the colon following them.

```
(function_definition
  declarator: (_) @func-declarator
  body: (_) @func-body)
```

It is also possible to capture a node that doesn't have a certain field, say, a `function_definition` without a `body` field:

```
(function_definition !body) @func-no-body
```

Quantify node

Tree-sitter recognizes quantification operators '`:*`', '`:+`', and '`:?`'. Their meanings are the same as in regular expressions: '`:*`' matches the preceding pattern zero or more times, '`:+`' matches one or more times, and '`:?`' matches zero or one times.

For example, the following pattern matches `type_declaration` nodes that have *zero or more* `long` keywords.

```
(type_declaration "long" :*) @long-type
```

The following pattern matches a type declaration that may or may not have a `long` keyword:

```
(type_declaration "long" :?) @long-type
```

Grouping

Similar to groups in regular expressions, we can bundle patterns into groups and apply quantification operators to them. For example, to express a comma-separated list of identifiers, one could write

```
(identifier) ("," (identifier)) :*
```

Alternation

Again, similar to regular expressions, we can express "match any one of these patterns" in a pattern. The syntax is a vector of patterns. For example, to capture some keywords in C, the pattern would be

```
[
  "return"
  "break"
  "if"
  "else"
] @keyword
```

Anchor

The anchor operator `:anchor` can be used to enforce juxtaposition, i.e., to enforce two things to be directly next to each other. The two "things" can be two nodes, or a child and the end of its parent. For example, to capture the first child, the last child, or two adjacent children:

```
;; Anchor the child with the end of its parent.
(compound_expression (_) @last-child :anchor)

;; Anchor the child with the beginning of its parent.
(compound_expression :anchor (_) @first-child)

;; Anchor two adjacent children.
(compound_expression
 (_) @prev-child
 :anchor
 (_) @next-child)
```

Note that the enforcement of juxtaposition ignores any anonymous nodes.

Predicate

It is possible to add predicate constraints to a pattern. For example, with the following pattern:

```
(
 (array :anchor (_) @first (_) @last :anchor)
 (:equal @first @last)
)
```

tree-sitter only matches arrays where the first element is equal to the last element. To attach a predicate to a pattern, we need to group them together. Currently there are three predicates: `:equal`, `:match`, and `:pred`.

`:equal` *arg1 arg2* [Predicate]
: Matches if *arg1* is equal to *arg2*. Arguments can be either strings or capture names. Capture names represent the text that the captured node spans in the buffer.

`:match` *regexp capture-name* [Predicate]
: Matches if the text that *capture-name*'s node spans in the buffer matches regular expression *regexp*, given as a string literal. Matching is case-sensitive.

`:pred` *fn* **&rest** *nodes* [Predicate]
: Matches if function *fn* returns non-`nil` when passed each node in *nodes* as arguments.

Note that a predicate can only refer to capture names that appear in the same pattern. Indeed, it makes little sense to refer to capture names in other patterns.

String patterns

Besides s-expressions, Emacs allows the tree-sitter's native query syntax to be used by writing them as strings. It largely resembles the s-expression syntax. For example, the following query

```
(treesit-query-capture
 node '((addition_expression
         left: (_) @left
         "+" @plus-sign
         right: (_) @right) @addition

         ["return" "break"] @keyword))
```

is equivalent to

```
(treesit-query-capture
 node "(addition_expression
        left: (_) @left
        \"+\" @plus-sign
        right: (_) @right) @addition

        [\"return\" \"break\"] @keyword")
```

Most patterns can be written directly as s-expressions inside a string. Only a few of them need modification:

- Anchor `:anchor` is written as '`.`'.
- '`:?`' is written as '`?`'.
- '`:*`' is written as '`*`'.

- ‘`:+`’ is written as ‘`+`’.
- `:equal`, `:match` and `:pred` are written as `#equal`, `#match` and `#pred`, respectively. In general, predicates change their ‘`:`’ to ‘`#`’.

For example,

```
'((
   (compound_expression :anchor (_) @first (_) :* @rest)
   (:match "love" @first)
   ))
```

is written in string form as

```
"(
  (compound_expression . (_) @first (_)* @rest)
  (#match \"love\" @first)
  )"
```

Compiling queries

If a query is intended to be used repeatedly, especially in tight loops, it is important to compile that query, because a compiled query is much faster than an uncompiled one. A compiled query can be used anywhere a query is accepted.

`treesit-query-compile` *language query* [Function]
: This function compiles *query* for *language* into a compiled query object and returns it.

  This function raises the `treesit-query-error` error if *query* is malformed. The signal data contains a description of the specific error. You can use `treesit-query-validate` to validate and debug the query.

`treesit-query-language` *query* [Function]
: This function returns the language of *query*.

`treesit-query-expand` *query* [Function]
: This function converts the s-expression *query* into the string format.

`treesit-pattern-expand` *pattern* [Function]
: This function converts the s-expression *pattern* into the string format.

For more details, read the tree-sitter project's documentation about pattern-matching, which can be found at `https://tree-sitter.github.io/tree-sitter/using-parsers#pattern-matching-with-queries`.

37.6 Parsing Text in Multiple Languages

Sometimes, the source of a programming language could contain snippets of other languages; HTML + CSS + JavaScript is one example. In that case, text segments written in different languages need to be assigned different parsers. Traditionally, this is achieved by using narrowing. While tree-sitter works with narrowing (see [tree-sitter narrowing], page 1047), the recommended way is instead to specify regions of buffer text (i.e., ranges) in which a

parser will operate. This section describes functions for setting and getting ranges for a parser.

Lisp programs should call `treesit-update-ranges` to make sure the ranges for each parser are correct before using parsers in a buffer, and call `treesit-language-at` to figure out the language responsible for the text at some position. These two functions don't work by themselves, they need major modes to set `treesit-range-settings` and `treesit-language-at-point-function`, which do the actual work. These functions and variables are explained in more detail towards the end of the section.

Getting and setting ranges

`treesit-parser-set-included-ranges` *parser ranges* [Function]

This function sets up *parser* to operate on *ranges*. The *parser* will only read the text of the specified ranges. Each range in *ranges* is a pair of the form `(beg . end)`.

The ranges in *ranges* must come in order and must not overlap. That is, in pseudo code:

```
(cl-loop for idx from 1 to (1- (length ranges))
         for prev = (nth (1- idx) ranges)
         for next = (nth idx ranges)
         should (<= (car prev) (cdr prev)
                    (car next) (cdr next)))
```

If *ranges* violates this constraint, or something else went wrong, this function signals the `treesit-range-invalid` error. The signal data contains a specific error message and the ranges we are trying to set.

This function can also be used for disabling ranges. If *ranges* is `nil`, the parser is set to parse the whole buffer.

Example:

```
(treesit-parser-set-included-ranges
 parser '((1 . 9) (16 . 24) (24 . 25)))
```

`treesit-parser-included-ranges` *parser* [Function]

This function returns the ranges set for *parser*. The return value is the same as the *ranges* argument of `treesit-parser-included-ranges`: a list of cons cells of the form `(beg . end)`. If *parser* doesn't have any ranges, the return value is `nil`.

```
(treesit-parser-included-ranges parser)
    ⇒ ((1 . 9) (16 . 24) (24 . 25))
```

`treesit-query-range` *source query* **&optional** *beg end* [Function]

This function matches *source* with *query* and returns the ranges of captured nodes. The return value is a list of cons cells of the form `(beg . end)`, where *beg* and *end* specify the beginning and the end of a region of text.

For convenience, *source* can be a language symbol, a parser, or a node. If it's a language symbol, this function matches in the root node of the first parser using that language; if a parser, this function matches in the root node of that parser; if a node, this function matches in that node.

The argument *query* is the query used to capture nodes (see Section 37.5 [Pattern Matching], page 1055). The capture names don't matter. The arguments *beg* and *end*, if both non-`nil`, limit the range in which this function queries.

Like other query functions, this function raises the `treesit-query-error` error if *query* is malformed.

Supporting multiple languages in Lisp programs

It should suffice for general Lisp programs to call the following two functions in order to support program sources that mix multiple languages.

`treesit-update-ranges` **&optional** *beg end* [Function]

This function updates ranges for parsers in the buffer. It makes sure the parsers' ranges are set correctly between *beg* and *end*, according to `treesit-range-settings`. If omitted, *beg* defaults to the beginning of the buffer, and *end* defaults to the end of the buffer.

For example, fontification functions use this function before querying for nodes in a region.

`treesit-language-at` *pos* [Function]

This function returns the language of the text at buffer position *pos*. Under the hood it calls `treesit-language-at-point-function` and returns its return value. If `treesit-language-at-point-function` is `nil`, this function returns the language of the first parser in the returned value of `treesit-parser-list`. If there is no parser in the buffer, it returns `nil`.

Supporting multiple languages in major modes

Normally, in a set of languages that can be mixed together, there is a *host language* and one or more *embedded languages*. A Lisp program usually first parses the whole document with the host language's parser, retrieves some information, sets ranges for the embedded languages with that information, and then parses the embedded languages.

Take a buffer containing HTML, CSS, and JavaScript as an example. A Lisp program will first parse the whole buffer with an HTML parser, then query the parser for `style_element` and `script_element` nodes, which correspond to CSS and JavaScript text, respectively. Then it sets the range of the CSS and JavaScript parsers to the range which their corresponding nodes span.

Given a simple HTML document:

```
<html>
  <script>1 + 2</script>
  <style>body { color: "blue"; }</style>
</html>
```

a Lisp program will first parse with a HTML parser, then set ranges for CSS and JavaScript parsers:

```
;; Create parsers.
(setq html (treesit-parser-create 'html))
(setq css (treesit-parser-create 'css))
(setq js (treesit-parser-create 'javascript))
```

```
;; Set CSS ranges.
(setq css-range
      (treesit-query-range
       'html
       '((style_element (raw_text) @capture))))
(treesit-parser-set-included-ranges css css-range)

;; Set JavaScript ranges.
(setq js-range
      (treesit-query-range
       'html
       '((script_element (raw_text) @capture))))
(treesit-parser-set-included-ranges js js-range)
```

Emacs automates this process in `treesit-update-ranges`. A multi-language major mode should set `treesit-range-settings` so that `treesit-update-ranges` knows how to perform this process automatically. Major modes should use the helper function `treesit-range-rules` to generate a value that can be assigned to `treesit-range-settings`. The settings in the following example directly translate into operations shown above.

```
(setq treesit-range-settings
      (treesit-range-rules
       :embed 'javascript
       :host 'html
       '((script_element (raw_text) @capture))

       :embed 'css
       :host 'html
       '((style_element (raw_text) @capture))))
```

`treesit-range-rules` **&rest** *query-specs* [Function]

This function is used to set `treesit-range-settings`. It takes care of compiling queries and other post-processing, and outputs a value that `treesit-range-settings` can have.

It takes a series of *query-spec*s, where each *query-spec* is a *query* preceded by zero or more *keyword*/*value* pairs. Each *query* is a tree-sitter query in either the string, s-expression, or compiled form, or a function.

If *query* is a tree-sitter query, it should be preceded by two *keyword*/*value* pairs, where the `:embed` keyword specifies the embedded language, and the `:host` keyword specifies the host language.

`treesit-update-ranges` uses *query* to figure out how to set the ranges for parsers for the embedded language. It queries *query* in a host language parser, computes the ranges which the captured nodes span, and applies these ranges to embedded language parsers.

If *query* is a function, it doesn't need any *keyword* and *value* pair. It should be a function that takes 2 arguments, *start* and *end*, and sets the ranges for parsers in the

current buffer in the region between *start* and *end*. It is fine for this function to set ranges in a larger region that encompasses the region between *start* and *end*.

`treesit-range-settings` [Variable]
This variable helps `treesit-update-ranges` in updating the ranges for parsers in the buffer. It is a list of *setting*s where the exact format of a *setting* is considered internal. You should use `treesit-range-rules` to generate a value that this variable can have.

`treesit-language-at-point-function` [Variable]
This variable's value should be a function that takes a single argument, *pos*, which is a buffer position, and returns the language of the buffer text at *pos*. This variable is used by `treesit-language-at`.

37.7 Developing major modes with tree-sitter

This section covers some general guidelines on developing tree-sitter integration for a major mode.

A major mode supporting tree-sitter features should roughly follow this pattern:

```
(define-derived-mode woomy-mode prog-mode "Woomy"
  "A mode for Woomy programming language."
  (when (treesit-ready-p 'woomy)
    (setq-local treesit-variables ...)
    ...
    (treesit-major-mode-setup)))
```

`treesit-ready-p` automatically emits a warning if conditions for enabling tree-sitter aren't met.

If a tree-sitter major mode shares setup with its "native" counterpart, one can create a "base mode" that contains the common setup, like this:

```
(define-derived-mode woomy--base-mode prog-mode "Woomy"
  "An internal mode for Woomy programming language."
  (common-setup)
  ...)

(define-derived-mode woomy-mode woomy--base-mode "Woomy"
  "A mode for Woomy programming language."
  (native-setup)
  ...)

(define-derived-mode woomy-ts-mode woomy--base-mode "Woomy"
  "A mode for Woomy programming language."
  (when (treesit-ready-p 'woomy)
    (setq-local treesit-variables ...)
    ...
    (treesit-major-mode-setup)))
```

`treesit-ready-p` *language* **&optional** *quiet* [Function]

This function checks for conditions for activating tree-sitter. It checks whether Emacs was built with tree-sitter, whether the buffer's size is not too large for tree-sitter to handle, and whether the grammar for *language* is available on the system (see Section 37.1 [Language Grammar], page 1042).

This function emits a warning if tree-sitter cannot be activated. If *quiet* is `message`, the warning is turned into a message; if *quiet* is `t`, no warning or message is displayed.

If all the necessary conditions are met, this function returns non-`nil`; otherwise it returns `nil`.

`treesit-major-mode-setup` [Function]

This function activates some tree-sitter features for a major mode.

Currently, it sets up the following features:

- If `treesit-font-lock-settings` (see Section 24.6.10 [Parser-based Font Lock], page 580) is non-`nil`, it sets up fontification.
- If `treesit-simple-indent-rules` (see Section 24.7.2 [Parser-based Indentation], page 592) is non-`nil`, it sets up indentation.
- If `treesit-defun-type-regexp` is non-`nil`, it sets up navigation functions for `beginning-of-defun` and `end-of-defun`.
- If `treesit-defun-name-function` is non-`nil`, it sets up add-log functions used by `add-log-current-defun`.
- If `treesit-simple-imenu-settings` (see Section 24.5 [Imenu], page 562) is non-`nil`, it sets up Imenu.

For more information on these built-in tree-sitter features, see Section 24.6.10 [Parser-based Font Lock], page 580, see Section 24.7.2 [Parser-based Indentation], page 592, and see Section 31.2.6 [List Motion], page 863.

For supporting mixing of multiple languages in a major mode, see Section 37.6 [Multiple Languages], page 1060.

Besides `beginning-of-defun` and `end-of-defun`, Emacs provides some additional functions for working with defuns: `treesit-defun-at-point` returns the defun node at point, and `treesit-defun-name` returns the name of a defun node.

`treesit-defun-at-point` [Function]

This function returns the defun node at point, or `nil` if none is found. It respects `treesit-defun-tactic`: if its value is `top-level`, this function returns the top-level defun, and if its value is `nested`, it returns the immediate enclosing defun.

This function requires `treesit-defun-type-regexp` to work. If it is `nil`, this function simply returns `nil`.

`treesit-defun-name` *node* [Function]

This function returns the defun name of *node*. It returns `nil` if there is no defun name for *node*, or if *node* is not a defun node, or if *node* is `nil`.

Depending on the language and major mode, the defun names are names like function name, class name, struct name, etc.

If `treesit-defun-name-function` is `nil`, this function always returns `nil`.

`treesit-defun-name-function` [Variable]

If non-`nil`, this variable's value should be a function that is called with a node as its argument, and returns the defun name of the node. The function should have the same semantics as `treesit-defun-name`: if the node is not a defun node, or the node is a defun node but doesn't have a name, or the node is `nil`, it should return `nil`.

37.8 Tree-sitter C API Correspondence

Emacs' tree-sitter integration doesn't expose every feature provided by tree-sitter's C API. Missing features include:

- Creating a tree cursor and navigating the syntax tree with it.
- Setting timeout and cancellation flag for a parser.
- Setting the logger for a parser.
- Printing a DOT graph of the syntax tree to a file.
- Copying and modifying a syntax tree. (Emacs doesn't expose a tree object.)
- Using (row, column) coordinates as position.
- Updating a node with changes. (In Emacs, retrieve a new node instead of updating the existing one.)
- Querying statics of a language grammar.

In addition, Emacs makes some changes to the C API to make the API more convenient and idiomatic:

- Instead of using byte positions, the Emacs Lisp API uses character positions.
- Null nodes are converted to `nil`.

Below is the correspondence between all C API functions and their ELisp counterparts. Sometimes one ELisp function corresponds to multiple C functions, and many C functions don't have an ELisp counterpart.

`ts_parser_new`	`treesit-parser-create`
`ts_parser_delete`	
`ts_parser_set_language`	
`ts_parser_language`	`treesit-parser-language`
`ts_parser_set_included_ranges`	`treesit-parser-set-included-ranges`
`ts_parser_included_ranges`	`treesit-parser-included-ranges`
`ts_parser_parse`	
`ts_parser_parse_string`	`treesit-parse-string`
`ts_parser_parse_string_encoding`	
`ts_parser_reset`	
`ts_parser_set_timeout_micros`	
`ts_parser_timeout_micros`	
`ts_parser_set_cancellation_flag`	
`ts_parser_cancellation_flag`	
`ts_parser_set_logger`	
`ts_parser_logger`	
`ts_parser_print_dot_graphs`	
`ts_tree_copy`	

```
ts_tree_delete
ts_tree_root_node
ts_tree_language
ts_tree_edit
ts_tree_get_changed_ranges
ts_tree_print_dot_graph
ts_node_type                              treesit-node-type
ts_node_symbol
ts_node_start_byte                        treesit-node-start
ts_node_start_point
ts_node_end_byte                          treesit-node-end
ts_node_end_point
ts_node_string                            treesit-node-string
ts_node_is_null
ts_node_is_named                          treesit-node-check
ts_node_is_missing                        treesit-node-check
ts_node_is_extra                          treesit-node-check
ts_node_has_changes
ts_node_has_error                         treesit-node-check
ts_node_parent                            treesit-node-parent
ts_node_child                             treesit-node-child
ts_node_field_name_for_child              treesit-node-field-name-for-child
ts_node_child_count                       treesit-node-child-count
ts_node_named_child                       treesit-node-child
ts_node_named_child_count                 treesit-node-child-count
ts_node_child_by_field_name               treesit-node-child-by-field-name
ts_node_child_by_field_id
ts_node_next_sibling                      treesit-node-next-sibling
ts_node_prev_sibling                      treesit-node-prev-sibling
ts_node_next_named_sibling                treesit-node-next-sibling
ts_node_prev_named_sibling                treesit-node-prev-sibling
ts_node_first_child_for_byte              treesit-node-first-child-for-pos
ts_node_first_named_child_for_byte        treesit-node-first-child-for-pos
ts_node_descendant_for_byte_range         treesit-node-descendant-for-range
ts_node_descendant_for_point_range
ts_node_named_descendant_for_byte_range   treesit-node-descendant-for-range
ts_node_named_descendant_for_point_range
ts_node_edit
ts_node_eq                                treesit-node-eq
ts_tree_cursor_new
ts_tree_cursor_delete
ts_tree_cursor_reset
ts_tree_cursor_current_node
ts_tree_cursor_current_field_name
ts_tree_cursor_current_field_id
ts_tree_cursor_goto_parent
ts_tree_cursor_goto_next_sibling
```

```
ts_tree_cursor_goto_first_child
ts_tree_cursor_goto_first_child_for_byte
ts_tree_cursor_goto_first_child_for_point
ts_tree_cursor_copy
ts_query_new
ts_query_delete
ts_query_pattern_count
ts_query_capture_count
ts_query_string_count
ts_query_start_byte_for_pattern
ts_query_predicates_for_pattern
ts_query_step_is_definite
ts_query_capture_name_for_id
ts_query_string_value_for_id
ts_query_disable_capture
ts_query_disable_pattern
ts_query_cursor_new
ts_query_cursor_delete
ts_query_cursor_exec                      treesit-query-capture
ts_query_cursor_did_exceed_match_limit
ts_query_cursor_match_limit
ts_query_cursor_set_match_limit
ts_query_cursor_set_byte_range
ts_query_cursor_set_point_range
ts_query_cursor_next_match
ts_query_cursor_remove_match
ts_query_cursor_next_capture
ts_language_symbol_count
ts_language_symbol_name
ts_language_symbol_for_name
ts_language_field_count
ts_language_field_name_for_id
ts_language_field_id_for_name
ts_language_symbol_type
ts_language_version
```

38 Abbrevs and Abbrev Expansion

An abbreviation or *abbrev* is a string of characters that may be expanded to a longer string. The user can insert the abbrev string and find it replaced automatically with the expansion of the abbrev. This saves typing.

The set of abbrevs currently in effect is recorded in an *abbrev table*. Each buffer has a local abbrev table, but normally all buffers in the same major mode share one abbrev table. There is also a global abbrev table. Normally both are used.

An abbrev table is represented as an obarray. See Section 9.3 [Creating Symbols], page 135, for information about obarrays. Each abbreviation is represented by a symbol in the obarray. The symbol's name is the abbreviation; its value is the expansion; its function definition is the hook function for performing the expansion (see Section 38.2 [Defining Abbrevs], page 1070); and its property list cell contains various additional properties, including the use count and the number of times the abbreviation has been expanded (see Section 38.6 [Abbrev Properties], page 1074).

Certain abbrevs, called *system abbrevs*, are defined by a major mode instead of the user. A system abbrev is identified by its non-`nil` `:system` property (see Section 38.6 [Abbrev Properties], page 1074). When abbrevs are saved to an abbrev file, system abbrevs are omitted. See Section 38.3 [Abbrev Files], page 1071.

Because the symbols used for abbrevs are not interned in the usual obarray, they will never appear as the result of reading a Lisp expression; in fact, normally they are never used except by the code that handles abbrevs. Therefore, it is safe to use them in a nonstandard way.

If the minor mode Abbrev mode is enabled, the buffer-local variable `abbrev-mode` is non-`nil`, and abbrevs are automatically expanded in the buffer. For the user-level commands for abbrevs, see Section "Abbrev Mode" in *The GNU Emacs Manual*.

38.1 Abbrev Tables

This section describes how to create and manipulate abbrev tables.

`make-abbrev-table` **&optional** *props* [Function]
: This function creates and returns a new, empty abbrev table—an obarray containing no symbols. It is a vector filled with zeros. *props* is a property list that is applied to the new table (see Section 38.7 [Abbrev Table Properties], page 1075).

`abbrev-table-p` *object* [Function]
: This function returns a non-`nil` value if *object* is an abbrev table.

`clear-abbrev-table` *abbrev-table* [Function]
: This function undefines all the abbrevs in *abbrev-table*, leaving it empty.

`copy-abbrev-table` *abbrev-table* [Function]
: This function returns a copy of *abbrev-table*—a new abbrev table containing the same abbrev definitions. It does *not* copy any property lists; only the names, values, and functions.

`define-abbrev-table` *tabname definitions* **&optional** *docstring* **&rest** *props* [Function]

This function defines *tabname* (a symbol) as an abbrev table name, i.e., as a variable whose value is an abbrev table. It defines abbrevs in the table according to *definitions*, a list of elements of the form `(abbrevname expansion [hook] [props...])`. These elements are passed as arguments to `define-abbrev`.

The optional string *docstring* is the documentation string of the variable *tabname*. The property list *props* is applied to the abbrev table (see Section 38.7 [Abbrev Table Properties], page 1075).

If this function is called more than once for the same *tabname*, subsequent calls add the definitions in *definitions* to *tabname*, rather than overwriting the entire original contents. (A subsequent call only overrides abbrevs explicitly redefined or undefined in *definitions*.)

`abbrev-table-name-list` [Variable]

This is a list of symbols whose values are abbrev tables. `define-abbrev-table` adds the new abbrev table name to this list.

`insert-abbrev-table-description` *name* **&optional** *human* [Function]

This function inserts before point a description of the abbrev table named *name*. The argument *name* is a symbol whose value is an abbrev table.

If *human* is non-`nil`, the description is human-oriented. System abbrevs are listed and identified as such. Otherwise the description is a Lisp expression—a call to `define-abbrev-table` that would define *name* as it is currently defined, but without the system abbrevs. (The mode or package using *name* is supposed to add these to *name* separately.)

38.2 Defining Abbrevs

`define-abbrev` is the low-level basic function for defining an abbrev in an abbrev table.

When a major mode defines a system abbrev, it should call `define-abbrev` and specify `t` for the `:system` property. Be aware that any saved non-system abbrevs are restored at startup, i.e., before some major modes are loaded. Therefore, major modes should not assume that their abbrev tables are empty when they are first loaded.

`define-abbrev` *abbrev-table name expansion* **&optional** *hook* **&rest** *props* [Function]

This function defines an abbrev named *name*, in *abbrev-table*, to expand to *expansion* and call *hook*, with properties *props* (see Section 38.6 [Abbrev Properties], page 1074). The return value is *name*. The `:system` property in *props* is treated specially here: if it has the value `force`, then it will overwrite an existing definition even for a non-system abbrev of the same name.

name should be a string. The argument *expansion* is normally the desired expansion (a string), or `nil` to undefine the abbrev. If it is anything but a string or `nil`, then the abbreviation expands solely by running *hook*.

The argument *hook* is a function or `nil`. If *hook* is non-`nil`, then it is called with no arguments after the abbrev is replaced with *expansion*; point is located at the end of *expansion* when *hook* is called.

If *hook* is a non-`nil` symbol whose `no-self-insert` property is non-`nil`, *hook* can explicitly control whether to insert the self-inserting input character that triggered the expansion. If *hook* returns non-`nil` in this case, that inhibits insertion of the character. By contrast, if *hook* returns `nil`, `expand-abbrev` (or `abbrev-insert`) also returns `nil`, as if expansion had not really occurred.

Normally, `define-abbrev` sets the variable `abbrevs-changed` to `t`, if it actually changes the abbrev. This is so that some commands will offer to save the abbrevs. It does not do this for a system abbrev, since those aren't saved anyway.

`only-global-abbrevs` [User Option]
If this variable is non-`nil`, it means that the user plans to use global abbrevs only. This tells the commands that define mode-specific abbrevs to define global ones instead. This variable does not alter the behavior of the functions in this section; it is examined by their callers.

38.3 Saving Abbrevs in Files

A file of saved abbrev definitions is actually a file of Lisp code. The abbrevs are saved in the form of a Lisp program to define the same abbrev tables with the same contents. Therefore, you can load the file with `load` (see Section 16.1 [How Programs Do Loading], page 299). However, the function `quietly-read-abbrev-file` is provided as a more convenient interface. Emacs automatically calls this function at startup.

User-level facilities such as `save-some-buffers` can save abbrevs in a file automatically, under the control of variables described here.

`abbrev-file-name` [User Option]
This is the default file name for reading and saving abbrevs. By default, Emacs will look for `~/.emacs.d/abbrev_defs`, and, if not found, for `~/.abbrev_defs`; if neither file exists, Emacs will create `~/.emacs.d/abbrev_defs`.

`quietly-read-abbrev-file` **&optional** *filename* [Function]
This function reads abbrev definitions from a file named *filename*, previously written with `write-abbrev-file`. If *filename* is omitted or `nil`, the file specified in `abbrev-file-name` is used.

As the name implies, this function does not display any messages.

`save-abbrevs` [User Option]
A non-`nil` value for `save-abbrevs` means that Emacs should offer to save abbrevs (if any have changed) when files are saved. If the value is `silently`, Emacs saves the abbrevs without asking the user. `abbrev-file-name` specifies the file to save the abbrevs in. The default value is `t`.

`abbrevs-changed` [Variable]
This variable is set non-`nil` by defining or altering any abbrevs (except system abbrevs). This serves as a flag for various Emacs commands to offer to save your abbrevs.

`write-abbrev-file` **&optional** *filename* [Command]
: Save all abbrev definitions (except system abbrevs), for all abbrev tables listed in `abbrev-table-name-list`, in the file *filename*, in the form of a Lisp program that when loaded will define the same abbrevs. Tables that do not have any abbrevs to save are omitted. If *filename* is `nil` or omitted, `abbrev-file-name` is used. This function returns `nil`.

38.4 Looking Up and Expanding Abbreviations

Abbrevs are usually expanded by certain interactive commands, including `self-insert-command`. This section describes the subroutines used in writing such commands, as well as the variables they use for communication.

`abbrev-symbol` *abbrev* **&optional** *table* [Function]
: This function returns the symbol representing the abbrev named *abbrev*. It returns `nil` if that abbrev is not defined. The optional second argument *table* is the abbrev table in which to look it up. If *table* is `nil`, this function tries first the current buffer's local abbrev table, and second the global abbrev table.

`abbrev-expansion` *abbrev* **&optional** *table* [Function]
: This function returns the string that *abbrev* would expand into (as defined by the abbrev tables used for the current buffer). It returns `nil` if *abbrev* is not a valid abbrev. The optional argument *table* specifies the abbrev table to use, as in `abbrev-symbol`.

`expand-abbrev` [Command]
: This command expands the abbrev before point, if any. If point does not follow an abbrev, this command does nothing. To do the expansion, it calls the function that is the value of the `abbrev-expand-function` variable, with no arguments, and returns whatever that function does.

  The default expansion function returns the abbrev symbol if it did expansion, and `nil` otherwise. If the abbrev symbol has a hook function that is a symbol whose `no-self-insert` property is non-`nil`, and if the hook function returns `nil` as its value, then the default expansion function returns `nil`, even though expansion did occur.

`abbrev-insert` *abbrev* **&optional** *name start end* [Function]
: This function inserts the abbrev expansion of `abbrev`, replacing the text between `start` and `end`. If `start` is omitted, it defaults to point. `name`, if non-`nil`, should be the name by which this abbrev was found (a string); it is used to figure out whether to adjust the capitalization of the expansion. The function returns `abbrev` if the abbrev was successfully inserted, otherwise it returns `nil`.

`abbrev-prefix-mark` **&optional** *arg* [Command]
: This command marks the current location of point as the beginning of an abbrev. The next call to `expand-abbrev` will use the text from here to point (where it is then) as the abbrev to expand, rather than using the previous word as usual.

  First, this command expands any abbrev before point, unless *arg* is non-`nil`. (Interactively, *arg* is the prefix argument.) Then it inserts a hyphen before point, to

indicate the start of the next abbrev to be expanded. The actual expansion removes the hyphen.

`abbrev-all-caps` [User Option]
When this is set non-`nil`, an abbrev entered entirely in upper case is expanded using all upper case. Otherwise, an abbrev entered entirely in upper case is expanded by capitalizing each word of the expansion.

`abbrev-start-location` [Variable]
The value of this variable is a buffer position (an integer or a marker) for `expand-abbrev` to use as the start of the next abbrev to be expanded. The value can also be `nil`, which means to use the word before point instead. `abbrev-start-location` is set to `nil` each time `expand-abbrev` is called. This variable is also set by `abbrev-prefix-mark`.

`abbrev-start-location-buffer` [Variable]
The value of this variable is the buffer for which `abbrev-start-location` has been set. Trying to expand an abbrev in any other buffer clears `abbrev-start-location`. This variable is set by `abbrev-prefix-mark`.

`last-abbrev` [Variable]
This is the `abbrev-symbol` of the most recent abbrev expanded. This information is left by `expand-abbrev` for the sake of the `unexpand-abbrev` command (see Section "Expanding Abbrevs" in *The GNU Emacs Manual*).

`last-abbrev-location` [Variable]
This is the location of the most recent abbrev expanded. This contains information left by `expand-abbrev` for the sake of the `unexpand-abbrev` command.

`last-abbrev-text` [Variable]
This is the exact expansion text of the most recent abbrev expanded, after case conversion (if any). Its value is `nil` if the abbrev has already been unexpanded. This contains information left by `expand-abbrev` for the sake of the `unexpand-abbrev` command.

`abbrev-expand-function` [Variable]
The value of this variable is a function that `expand-abbrev` will call with no arguments to do the expansion. The function can do anything it wants before and after performing the expansion. It should return the abbrev symbol if expansion took place.

The following sample code shows a simple use of `abbrev-expand-function`. It assumes that `foo-mode` is a mode for editing certain files in which lines that start with '`#`' are comments. You want to use Text mode abbrevs for those lines. The regular local abbrev table, `foo-mode-abbrev-table` is appropriate for all other lines. See Section 38.5 [Standard Abbrev Tables], page 1074, for the definitions of `local-abbrev-table` and `text-mode-abbrev-table`. See Section 13.12 [Advising Functions], page 252, for details of `add-function`.

```
(defun foo-mode-abbrev-expand-function (expand)
  (if (not (save-excursion (forward-line 0) (eq (char-after) ?#)))
      ;; Performs normal expansion.
```

```
      (funcall expand)
    ;; We're inside a comment: use the text-mode abbrevs.
    (let ((local-abbrev-table text-mode-abbrev-table))
      (funcall expand))))

(add-hook 'foo-mode-hook
          (lambda ()
            (add-function :around (local 'abbrev-expand-function)
                          #'foo-mode-abbrev-expand-function)))
```

38.5 Standard Abbrev Tables

Here we list the variables that hold the abbrev tables for the preloaded major modes of Emacs.

`global-abbrev-table` [Variable]
: This is the abbrev table for mode-independent abbrevs. The abbrevs defined in it apply to all buffers. Each buffer may also have a local abbrev table, whose abbrev definitions take precedence over those in the global table.

`local-abbrev-table` [Variable]
: The value of this buffer-local variable is the (mode-specific) abbreviation table of the current buffer. It can also be a list of such tables.

`abbrev-minor-mode-table-alist` [Variable]
: The value of this variable is a list of elements of the form `(`*`mode . abbrev-table`*`)` where *mode* is the name of a variable: if the variable is bound to a non-`nil` value, then the *abbrev-table* is active, otherwise it is ignored. *abbrev-table* can also be a list of abbrev tables.

`fundamental-mode-abbrev-table` [Variable]
: This is the local abbrev table used in Fundamental mode; in other words, it is the local abbrev table in all buffers in Fundamental mode.

`text-mode-abbrev-table` [Variable]
: This is the local abbrev table used in Text mode.

`lisp-mode-abbrev-table` [Variable]
: This is the local abbrev table used in Lisp mode. It is the parent of the local abbrev table used in Emacs Lisp mode. See Section 38.7 [Abbrev Table Properties], page 1075.

38.6 Abbrev Properties

Abbrevs have properties, some of which influence the way they work. You can provide them as arguments to `define-abbrev`, and manipulate them with the following functions:

`abbrev-put` *abbrev prop val* [Function]
: Set the property *prop* of *abbrev* to value *val*.

`abbrev-get` *abbrev prop* [Function]
: Return the property *prop* of *abbrev*, or `nil` if the abbrev has no such property.

The following properties have special meanings:

`:count` This property counts the number of times the abbrev has been expanded. If not explicitly set, it is initialized to 0 by `define-abbrev`.

`:system` If non-`nil`, this property marks the abbrev as a system abbrev. Such abbrevs are not saved (see Section 38.3 [Abbrev Files], page 1071).

`:enable-function`
If non-`nil`, this property should be a function of no arguments which returns `nil` if the abbrev should not be used and `t` otherwise.

`:case-fixed`
If non-`nil`, this property indicates that the case of the abbrev's name is significant and should only match a text with the same pattern of capitalization. It also disables the code that modifies the capitalization of the expansion.

38.7 Abbrev Table Properties

Like abbrevs, abbrev tables have properties, some of which influence the way they work. You can provide them as arguments to `define-abbrev-table`, and manipulate them with the functions:

`abbrev-table-put` *table prop val* [Function]
Set the property *prop* of abbrev table *table* to value *val*.

`abbrev-table-get` *table prop* [Function]
Return the property *prop* of abbrev table *table*, or `nil` if *table* has no such property.

The following properties have special meaning:

`:enable-function`
This is like the `:enable-function` abbrev property except that it applies to all abbrevs in the table. It is used before even trying to find the abbrev before point, so it can dynamically modify the abbrev table.

`:case-fixed`
This is like the `:case-fixed` abbrev property except that it applies to all abbrevs in the table.

`:regexp` If non-`nil`, this property is a regular expression that indicates how to extract the name of the abbrev before point, before looking it up in the table. When the regular expression matches before point, the abbrev name is expected to be in submatch 1. If this property is `nil`, the default is to use `backward-word` and `forward-word` to find the name. This property allows the use of abbrevs whose name contains characters of non-word syntax.

`:parents` This property holds a list of tables from which to inherit other abbrevs.

`:abbrev-table-modiff`
This property holds a counter incremented each time a new abbrev is added to the table.

39 Threads

Emacs Lisp provides a limited form of concurrency, called *threads*. All the threads in a given instance of Emacs share the same memory. Concurrency in Emacs Lisp is "mostly cooperative", meaning that Emacs will only switch execution between threads at well-defined times. However, the Emacs thread support has been designed in a way to later allow more fine-grained concurrency, and correct programs should not rely on cooperative threading.

Currently, thread switching will occur upon explicit request via `thread-yield`, when waiting for keyboard input or for process output from asynchronous processes (e.g., during `accept-process-output`), or during blocking operations relating to threads, such as mutex locking or `thread-join`.

Emacs Lisp provides primitives to create and control threads, and also to create and control mutexes and condition variables, useful for thread synchronization.

While global variables are shared among all Emacs Lisp threads, local variables are not—a dynamic `let` binding is local. Each thread also has its own current buffer (see Section 28.2 [Current Buffer], page 673) and its own match data (see Section 35.6 [Match Data], page 1014).

Note that `let` bindings are treated specially by the Emacs Lisp implementation. There is no way to duplicate this unwinding and rewinding behavior other than by using `let`. For example, a manual implementation of `let` written using `unwind-protect` cannot arrange for variable values to be thread-specific.

In the case of lexical bindings (see Section 12.10 [Variable Scoping], page 201), a closure is an object like any other in Emacs Lisp, and bindings in a closure are shared by any threads invoking the closure.

39.1 Basic Thread Functions

Threads can be created and waited for. A thread cannot be exited directly, but the current thread can be exited implicitly, and other threads can be signaled.

`make-thread` *function* **&optional** *name* [Function]

Create a new thread of execution which invokes *function*. When *function* returns, the thread exits.

The new thread is created with no local variable bindings in effect. The new thread's current buffer is inherited from the current thread.

name can be supplied to give a name to the thread. The name is used for debugging and informational purposes only; it has no meaning to Emacs. If *name* is provided, it must be a string.

This function returns the new thread.

`threadp` *object* [Function]

This function returns `t` if *object* represents an Emacs thread, `nil` otherwise.

`thread-join` *thread* [Function]

Block until *thread* exits, or until the current thread is signaled. It returns the result of the *thread* function. If *thread* has already exited, this returns immediately.

`thread-signal` *thread error-symbol data* [Function]
Like `signal` (see Section 11.7.3.1 [Signaling Errors], page 180), but the signal is delivered in the thread *thread*. If *thread* is the current thread, then this just calls `signal` immediately. Otherwise, *thread* will receive the signal as soon as it becomes current. If *thread* was blocked by a call to `mutex-lock`, `condition-wait`, or `thread-join`; `thread-signal` will unblock it.

If *thread* is the main thread, the signal is not propagated there. Instead, it is shown as message in the main thread.

`thread-yield` [Function]
Yield execution to the next runnable thread.

`thread-name` *thread* [Function]
Return the name of *thread*, as specified to `make-thread`.

`thread-live-p` *thread* [Function]
Return `t` if *thread* is alive, or `nil` if it is not. A thread is alive as long as its function is still executing.

`thread--blocker` *thread* [Function]
Return the object that *thread* is waiting on. This function is primarily intended for debugging, and is given a "double hyphen" name to indicate that.

If *thread* is blocked in `thread-join`, this returns the thread for which it is waiting.

If *thread* is blocked in `mutex-lock`, this returns the mutex.

If *thread* is blocked in `condition-wait`, this returns the condition variable.

Otherwise, this returns `nil`.

`current-thread` [Function]
Return the current thread.

`all-threads` [Function]
Return a list of all the live thread objects. A new list is returned by each invocation.

`main-thread` [Variable]
This variable keeps the main thread Emacs is running, or `nil` if Emacs is compiled without thread support.

When code run by a thread signals an error that is unhandled, the thread exits. Other threads can access the error form which caused the thread to exit using the following function.

`thread-last-error` **&optional** *cleanup* [Function]
This function returns the last error form recorded when a thread exited due to an error. Each thread that exits abnormally overwrites the form stored by the previous thread's error with a new value, so only the last one can be accessed. If *cleanup* is non-`nil`, the stored form is reset to `nil`.

39.2 Mutexes

A *mutex* is an exclusive lock. At any moment, zero or one threads may own a mutex. If a thread attempts to acquire a mutex, and the mutex is already owned by some other thread, then the acquiring thread will block until the mutex becomes available.

Emacs Lisp mutexes are of a type called *recursive*, which means that a thread can re-acquire a mutex it owns any number of times. A mutex keeps a count of how many times it has been acquired, and each acquisition of a mutex must be paired with a release. The last release by a thread of a mutex reverts it to the unowned state, potentially allowing another thread to acquire the mutex.

`mutexp` *object* [Function]
: This function returns `t` if *object* represents an Emacs mutex, `nil` otherwise.

`make-mutex` **&optional** *name* [Function]
: Create a new mutex and return it. If *name* is specified, it is a name given to the mutex. It must be a string. The name is for debugging purposes only; it has no meaning to Emacs.

`mutex-name` *mutex* [Function]
: Return the name of *mutex*, as specified to `make-mutex`.

`mutex-lock` *mutex* [Function]
: This will block until this thread acquires *mutex*, or until this thread is signaled using `thread-signal`. If *mutex* is already owned by this thread, this simply returns.

`mutex-unlock` *mutex* [Function]
: Release *mutex*. If *mutex* is not owned by this thread, this will signal an error.

`with-mutex` *mutex body...* [Macro]
: This macro is the simplest and safest way to evaluate forms while holding a mutex. It acquires *mutex*, invokes *body*, and then releases *mutex*. It returns the result of *body*.

39.3 Condition Variables

A *condition variable* is a way for a thread to block until some event occurs. A thread can wait on a condition variable, to be woken up when some other thread notifies the condition.

A condition variable is associated with a mutex and, conceptually, with some condition. For proper operation, the mutex must be acquired, and then a waiting thread must loop, testing the condition and waiting on the condition variable. For example:

```
(with-mutex mutex
  (while (not global-variable)
    (condition-wait cond-var)))
```

The mutex ensures atomicity, and the loop is for robustness—there may be spurious notifications.

Similarly, the mutex must be held before notifying the condition. The typical, and best, approach is to acquire the mutex, make the changes associated with this condition, and then notify it:

```
(with-mutex mutex
```

```
(setq global-variable (some-computation))
(condition-notify cond-var))
```

`make-condition-variable` *mutex* **&optional** *name* [Function]
Make a new condition variable associated with *mutex*. If *name* is specified, it is a name given to the condition variable. It must be a string. The name is for debugging purposes only; it has no meaning to Emacs.

`condition-variable-p` *object* [Function]
This function returns `t` if *object* represents a condition variable, `nil` otherwise.

`condition-wait` *cond* [Function]
Wait for another thread to notify *cond*, a condition variable. This function will block until the condition is notified, or until a signal is delivered to this thread using `thread-signal`.

It is an error to call `condition-wait` without holding the condition's associated mutex.

`condition-wait` releases the associated mutex while waiting. This allows other threads to acquire the mutex in order to notify the condition.

`condition-notify` *cond* **&optional** *all* [Function]
Notify *cond*. The mutex with *cond* must be held before calling this. Ordinarily a single waiting thread is woken by `condition-notify`; but if *all* is not `nil`, then all threads waiting on *cond* are notified.

`condition-notify` releases the associated mutex while waiting. This allows other threads to acquire the mutex in order to wait on the condition.

`condition-name` *cond* [Function]
Return the name of *cond*, as passed to `make-condition-variable`.

`condition-mutex` *cond* [Function]
Return the mutex associated with *cond*. Note that the associated mutex cannot be changed.

39.4 The Thread List

The `list-threads` command lists all the currently alive threads. In the resulting buffer, each thread is identified either by the name passed to `make-thread` (see Section 39.1 [Basic Thread Functions], page 1076), or by its unique internal identifier if it was not created with a name. The status of each thread at the time of the creation or last update of the buffer is shown, in addition to the object the thread was blocked on at the time, if it was blocked.

`thread-list-refresh-seconds` [Variable]
The `*Threads*` buffer will automatically update twice per second. You can make the refresh rate faster or slower by customizing this variable.

Here are the commands available in the thread list buffer:

`b` Show a backtrace of the thread at point. This will show where in its code the thread had yielded or was blocked at the moment you pressed `b`. Be aware

that the backtrace is a snapshot; the thread could have meanwhile resumed execution, and be in a different state, or could have exited.

You may use `g` in the thread's backtrace buffer to get an updated backtrace, as backtrace buffers do not automatically update. See Section 19.1.8 [Backtraces], page 339, for a description of backtraces and the other commands which work on them.

`s` Signal the thread at point. After `s`, type `q` to send a quit signal or `e` to send an error signal. Threads may implement handling of signals, but the default behavior is to exit on any signal. Therefore you should only use this command if you understand how to restart the target thread, because your Emacs session may behave incorrectly if necessary threads are killed.

`g` Update the list of threads and their statuses.

40 Processes

In the terminology of operating systems, a *process* is a space in which a program can execute. Emacs runs in a process. Emacs Lisp programs can invoke other programs in processes of their own. These are called *subprocesses* or *child processes* of the Emacs process, which is their *parent process*.

A subprocess of Emacs may be *synchronous* or *asynchronous*, depending on how it is created. When you create a synchronous subprocess, the Lisp program waits for the subprocess to terminate before continuing execution. When you create an asynchronous subprocess, it can run in parallel with the Lisp program. This kind of subprocess is represented within Emacs by a Lisp object which is also called a "process". Lisp programs can use this object to communicate with the subprocess or to control it. For example, you can send signals, obtain status information, receive output from the process, or send input to it.

In addition to processes that run programs, Lisp programs can open connections of several types to devices or processes running on the same machine or on other machines. The supported connection types are: TCP and UDP network connections, serial port connections, and pipe connections. Each such connection is also represented by a process object.

`processp` *object* [Function]
: This function returns `t` if *object* represents an Emacs process object, `nil` otherwise. The process object can represent a subprocess running a program or a connection of any supported type.

In addition to subprocesses of the current Emacs session, you can also access other processes running on your machine. See Section 40.12 [System Processes], page 1111.

40.1 Functions that Create Subprocesses

There are three primitives that create a new subprocess in which to run a program. One of them, `make-process`, creates an asynchronous process and returns a process object (see Section 40.4 [Asynchronous Processes], page 1089). The other two, `call-process` and `call-process-region`, create a synchronous process and do not return a process object (see Section 40.3 [Synchronous Processes], page 1084). There are various higher-level functions that make use of these primitives to run particular types of process.

Synchronous and asynchronous processes are explained in the following sections. Since the three functions are all called in a similar fashion, their common arguments are described here.

In all cases, the functions specify the program to be run. An error is signaled if the file is not found or cannot be executed. If the file name is relative, the variable `exec-path` contains a list of directories to search. Emacs initializes `exec-path` when it starts up, based on the value of the environment variable `PATH`. The standard file name constructs, '`~`', '`.`', and '`..`', are interpreted as usual in `exec-path`, but environment variable substitutions ('`$HOME`', etc.) are not recognized; use `substitute-in-file-name` to perform them (see Section 26.9.4 [File Name Expansion], page 641). `nil` in this list refers to `default-directory`.

Executing a program can also try adding suffixes to the specified name:

`exec-suffixes` [User Option]

This variable is a list of suffixes (strings) to try adding to the specified program file name. The list should include `""` if you want the name to be tried exactly as specified. The default value is system-dependent.

Please note: The argument *program* contains only the name of the program file; it may not contain any command-line arguments. You must use a separate argument, *args*, to provide those, as described below.

Each of the subprocess-creating functions has a *buffer-or-name* argument that specifies where the output from the program will go. It should be a buffer or a buffer name; if it is a buffer name, that will create the buffer if it does not already exist. It can also be `nil`, which says to discard the output, unless a custom filter function handles it. (See Section 40.9.2 [Filter Functions], page 1104, and Chapter 20 [Read and Print], page 372.) Normally, you should avoid having multiple processes send output to the same buffer because their output would be intermixed randomly. For synchronous processes, you can send the output to a file instead of a buffer (and the corresponding argument is therefore more appropriately called *destination*). By default, both standard output and standard error streams go to the same destination, but all the 3 primitives allow optionally to direct the standard error stream to a different destination.

All three of the subprocess-creating functions allow to specify command-line arguments for the process to run. For `call-process` and `call-process-region`, these come in the form of a `&rest` argument, *args*. For `make-process`, both the program to run and its command-line arguments are specified as a list of strings. The command-line arguments must all be strings, and they are supplied to the program as separate argument strings. Wildcard characters and other shell constructs have no special meanings in these strings, since the strings are passed directly to the specified program.

The subprocess inherits its environment from Emacs, but you can specify overrides for it with `process-environment`. See Section 42.3 [System Environment], page 1270. The subprocess gets its current directory from the value of `default-directory`.

`exec-directory` [Variable]

The value of this variable is a string, the name of a directory that contains programs that come with GNU Emacs and are intended for Emacs to invoke. The program `movemail` is an example of such a program; Rmail uses it to fetch new mail from an inbox.

`exec-path` [User Option]

The value of this variable is a list of directories to search for programs to run in subprocesses. Each element is either the name of a directory (i.e., a string), or `nil`, which stands for the default directory (which is the value of `default-directory`). See Section 26.6.6 [Locating Files], page 630, for the details of this search.

The value of `exec-path` is used by `call-process` and `start-process` when the *program* argument is not an absolute file name.

Generally, you should not modify `exec-path` directly. Instead, ensure that your `PATH` environment variable is set appropriately before starting Emacs. Trying to modify `exec-path` independently of `PATH` can lead to confusing results.

`exec-path` [Function]

This function is an extension of the variable `exec-path`. If `default-directory` indicates a remote directory, this function returns a list of directories used for searching programs on the respective remote host. In case of a local `default-directory`, the function returns just the value of the variable `exec-path`.

40.2 Shell Arguments

Lisp programs sometimes need to run a shell and give it a command that contains file names that were specified by the user. These programs ought to be able to support any valid file name. But the shell gives special treatment to certain characters, and if these characters occur in the file name, they will confuse the shell. To handle these characters, use the function `shell-quote-argument`:

`shell-quote-argument` *argument* **&optional** *posix* [Function]

This function returns a string that represents, in shell syntax, an argument whose actual contents are *argument*. It should work reliably to concatenate the return value into a shell command and then pass it to a shell for execution.

Precisely what this function does depends on your operating system. The function is designed to work with the syntax of your system's standard shell; if you use an unusual shell, you will need to redefine this function. See Section 42.22 [Security Considerations], page 1305.

```
;; This example shows the behavior on GNU and Unix systems.
(shell-quote-argument "foo > bar")
     ⇒ "foo\\ \\>\\ bar"


;; This example shows the behavior on MS-DOS and MS-Windows.
(shell-quote-argument "foo > bar")
     ⇒ "\"foo > bar\""
```

Here's an example of using `shell-quote-argument` to construct a shell command:

```
(concat "diff -u "
        (shell-quote-argument oldfile)
        " "
        (shell-quote-argument newfile))
```

If the optional *posix* argument is non-`nil`, *argument* is quoted according to POSIX shell quoting rules, regardless of the system's shell. This is useful when your shell could run on a remote host, which requires a POSIX shell in general.

```
(shell-quote-argument "foo > bar" (file-remote-p default-directory))
```

The following two functions are useful for combining a list of individual command-line argument strings into a single string, and taking a string apart into a list of individual command-line arguments. These functions are mainly intended for converting user input in the minibuffer, a Lisp string, into a list of string arguments to be passed to `make-process`, `call-process` or `start-process`, or for converting such lists of arguments into a single Lisp string to be presented in the minibuffer or echo area. Note that if a shell is involved (e.g., if using `call-process-shell-command`), arguments should still be protected

by `shell-quote-argument`; `combine-and-quote-strings` is *not* intended to protect special characters from shell evaluation.

`split-string-shell-command` *string* [Function]
: This function splits *string* into substrings, respecting double and single quotes, as well as backslash quoting.

```
(split-string-shell-command "ls /tmp/'foo bar'")
     ⇒ ("ls" "/tmp/foo bar")
```

`split-string-and-unquote` *string* **&optional** *separators* [Function]
: This function splits *string* into substrings at matches for the regular expression *separators*, like `split-string` does (see Section 4.3 [Creating Strings], page 55); in addition, it removes quoting from the substrings. It then makes a list of the substrings and returns it.

  If *separators* is omitted or `nil`, it defaults to `"\\s-+"`, which is a regular expression that matches one or more characters with whitespace syntax (see Section 36.2.1 [Syntax Class Table], page 1026).

  This function supports two types of quoting: enclosing a whole string in double quotes `"..."`, and quoting individual characters with a backslash escape '\'. The latter is also used in Lisp strings, so this function can handle those as well.

`combine-and-quote-strings` *list-of-strings* **&optional** *separator* [Function]
: This function concatenates *list-of-strings* into a single string, quoting each string as necessary. It also sticks the *separator* string between each pair of strings; if *separator* is omitted or `nil`, it defaults to `" "`. The return value is the resulting string.

  The strings in *list-of-strings* that need quoting are those that include *separator* as their substring. Quoting a string encloses it in double quotes `"..."`. In the simplest case, if you are consing a command from the individual command-line arguments, every argument that includes embedded blanks will be quoted.

40.3 Creating a Synchronous Process

After a *synchronous process* is created, Emacs waits for the process to terminate before continuing. Starting Dired on GNU or Unix[1] is an example of this: it runs `ls` in a synchronous process, then modifies the output slightly. Because the process is synchronous, the entire directory listing arrives in the buffer before Emacs tries to do anything with it.

While Emacs waits for the synchronous subprocess to terminate, the user can quit by typing *C-g*. The first *C-g* tries to kill the subprocess with a `SIGINT` signal; but it waits until the subprocess actually terminates before quitting. If during that time the user types another *C-g*, that kills the subprocess instantly with `SIGKILL` and quits immediately (except on MS-DOS, where killing other processes doesn't work). See Section 22.11 [Quitting], page 472.

The synchronous subprocess functions return an indication of how the process terminated.

[1] On other systems, Emacs uses a Lisp emulation of `ls`; see Section 26.10 [Contents of Directories], page 648.

The output from a synchronous subprocess is generally decoded using a coding system, much like text read from a file. The input sent to a subprocess by `call-process-region` is encoded using a coding system, much like text written into a file. See Section 34.10 [Coding Systems], page 980.

`call-process` *program* **&optional** *infile destination display* **&rest** *args* [Function]

This function calls *program* and waits for it to finish.

The current working directory of the subprocess is set to the current buffer's value of `default-directory` if that is local (as determined by `unhandled-file-name-directory`), or `"~"` otherwise. If you want to run a process in a remote directory use `process-file`.

The standard input for the new process comes from file *infile* if *infile* is not `nil`, and from the null device otherwise. The argument *destination* says where to put the process output. Here are the possibilities:

a buffer
: Insert the output in that buffer, before point. This includes both the standard output stream and the standard error stream of the process.

a buffer name (a string)
: Insert the output in a buffer with that name, before point.

`t`
: Insert the output in the current buffer, before point.

`nil`
: Discard the output.

0
: Discard the output, and return `nil` immediately without waiting for the subprocess to finish.

  In this case, the process is not truly synchronous, since it can run in parallel with Emacs; but you can think of it as synchronous in that Emacs is essentially finished with the subprocess as soon as this function returns.

  MS-DOS doesn't support asynchronous subprocesses, so this option doesn't work there.

`(:file` *file-name*`)`
: Send the output to the file name specified, overwriting it if it already exists.

`(`*real-destination error-destination*`)`
: Keep the standard output stream separate from the standard error stream; deal with the ordinary output as specified by *real-destination*, and dispose of the error output according to *error-destination*. If *error-destination* is `nil`, that means to discard the error output, `t` means mix it with the ordinary output, and a string specifies a file name to redirect error output into.

  You can't directly specify a buffer to put the error output in; that is too difficult to implement. But you can achieve this result by sending the error output to a temporary file and then inserting the file into a buffer when the subprocess finishes.

If *display* is non-`nil`, then `call-process` redisplays the buffer as output is inserted. (However, if the coding system chosen for decoding output is `undecided`, meaning deduce the encoding from the actual data, then redisplay sometimes cannot continue once non-ASCII characters are encountered. There are fundamental reasons why it is hard to fix this; see Section 40.9 [Output from Processes], page 1102.)

Otherwise the function `call-process` does no redisplay, and the results become visible on the screen only when Emacs redisplays that buffer in the normal course of events.

The remaining arguments, *args*, are strings that specify command line arguments for the program. Each string is passed to *program* as a separate argument.

The value returned by `call-process` (unless you told it not to wait) indicates the reason for process termination. A number gives the exit status of the subprocess; 0 means success, and any other value means failure. If the process terminated with a signal, `call-process` returns a string describing the signal. If you told `call-process` not to wait, it returns `nil`.

In the examples below, the buffer '`foo`' is current.

```
(call-process "pwd" nil t)
     ⇒ 0

---------- Buffer: foo ----------
/home/lewis/manual
---------- Buffer: foo ----------

(call-process "grep" nil "bar" nil "lewis" "/etc/passwd")
     ⇒ 0

---------- Buffer: bar ----------
lewis:x:1001:1001:Bil Lewis,,,,:/home/lewis:/bin/bash

---------- Buffer: bar ----------
```

Here is an example of the use of `call-process`, as used to be found in the definition of the `insert-directory` function:

```
(call-process insert-directory-program nil t nil switches
              (if full-directory-p
                  (concat (file-name-as-directory file) ".")
                file))
```

`process-file` *program* **&optional** *infile buffer display* **&rest** *args* [Function]

This function processes files synchronously in a separate process. It is similar to `call-process`, but may invoke a file name handler based on the value of the variable `default-directory`, which specifies the current working directory of the subprocess.

The arguments are handled in almost the same way as for `call-process`, with the following differences:

Some file name handlers may not support all combinations and forms of the arguments *infile*, *buffer*, and *display*. For example, some file name handlers might behave as if *display* were `nil`, regardless of the value actually passed. As another example, some file name handlers might not support separating standard output and error output by way of the *buffer* argument.

If a file name handler is invoked, it determines the program to run based on the first argument *program*. For instance, suppose that a handler for remote files is invoked. Then the path that is used for searching for the program might be different from `exec-path`.

The second argument *infile* may invoke a file name handler. The file name handler could be different from the handler chosen for the `process-file` function itself. (For example, `default-directory` could be on one remote host, and *infile* on a different remote host. Or `default-directory` could be non-special, whereas *infile* is on a remote host.)

If *buffer* is a list of the form `(real-destination error-destination)`, and *error-destination* names a file, then the same remarks as for *infile* apply.

The remaining arguments (*args*) will be passed to the process verbatim. Emacs is not involved in processing file names that are present in *args*. To avoid confusion, it may be best to avoid absolute file names in *args*, but rather to specify all file names as relative to `default-directory`. The function `file-relative-name` is useful for constructing such relative file names. Alternatively, you can use `file-local-name` (see Section 26.12 [Magic File Names], page 651) to obtain an absolute file name as seen from the remote host's perspective.

`process-file-side-effects` [Variable]

This variable indicates whether a call of `process-file` changes remote files.

By default, this variable is always set to `t`, meaning that a call of `process-file` could potentially change any file on a remote host. When set to `nil`, a file name handler could optimize its behavior with respect to remote file attribute caching.

You should only ever change this variable with a let-binding; never with `setq`.

`process-file-return-signal-string` [User Option]

This user option indicates whether a call of `process-file` returns a string describing the signal interrupting a remote process.

When a process returns an exit code greater than 128, it is interpreted as a signal. `process-file` requires to return a string describing this signal.

Since there are processes violating this rule, returning exit codes greater than 128 which are not bound to a signal, `process-file` returns always the exit code as natural number for remote processes. Setting this user option to non-`nil` forces `process-file` to interpret such exit codes as signals, and to return a corresponding string.

`call-process-region` *start end program* **&optional** *delete destination display* **&rest** *args* [Function]

This function sends the text from *start* to *end* as standard input to a process running *program*. It deletes the text sent if *delete* is non-`nil`; this is useful when *destination* is `t`, to insert the output in the current buffer in place of the input.

The arguments *destination* and *display* control what to do with the output from the subprocess, and whether to update the display as it comes in. For details, see the description of `call-process`, above. If *destination* is the integer 0, `call-process-region` discards the output and returns `nil` immediately, without waiting for the

subprocess to finish (this only works if asynchronous subprocesses are supported; i.e., not on MS-DOS).

The remaining arguments, *args*, are strings that specify command line arguments for the program.

The return value of `call-process-region` is just like that of `call-process`: `nil` if you told it to return without waiting; otherwise, a number or string which indicates how the subprocess terminated.

In the following example, we use `call-process-region` to run the `cat` utility, with standard input being the first five characters in buffer '`foo`' (the word '`input`'). `cat` copies its standard input into its standard output. Since the argument *destination* is `t`, this output is inserted in the current buffer.

```
---------- Buffer: foo ----------
input⋆
---------- Buffer: foo ----------

(call-process-region 1 6 "cat" nil t)
     ⇒ 0

---------- Buffer: foo ----------
inputinput⋆
---------- Buffer: foo ----------
```

For example, the `shell-command-on-region` command uses `call-shell-region` in a manner similar to this:

```
(call-shell-region
 start end
 command                ; shell command
 nil                    ; do not delete region
 buffer)                ; send output to buffer
```

`call-process-shell-command` *command* **&optional** *infile destination display* [Function]

This function executes the shell command *command* synchronously. The other arguments are handled as in `call-process`. An old calling convention allowed passing any number of additional arguments after *display*, which were concatenated to *command*; this is still supported, but strongly discouraged.

`process-file-shell-command` *command* **&optional** *infile destination display* [Function]

This function is like `call-process-shell-command`, but uses `process-file` internally. Depending on `default-directory`, *command* can be executed also on remote hosts. An old calling convention allowed passing any number of additional arguments after *display*, which were concatenated to *command*; this is still supported, but strongly discouraged.

`call-shell-region` *start end command* **&optional** *delete destination* [Function]

This function sends the text from *start* to *end* as standard input to an inferior shell running *command*. This function is similar than `call-process-region`, with process being a shell. The arguments `delete`, `destination` and the return value are like in `call-process-region`. Note that this function doesn't accept additional arguments.

`shell-command-to-string` *command* [Function]
: This function executes *command* (a string) as a shell command, then returns the command's output as a string.

`process-lines` *program* **&rest** *args* [Function]
: This function runs *program*, waits for it to finish, and returns its output as a list of strings. Each string in the list holds a single line of text output by the program; the end-of-line characters are stripped from each line. The arguments beyond *program*, *args*, are strings that specify command-line arguments with which to run the program.

  If *program* exits with a non-zero exit status, this function signals an error.

  This function works by calling `call-process`, so program output is decoded in the same way as for `call-process`.

`process-lines-ignore-status` *program* **&rest** *args* [Function]
: This function is just like `process-lines`, but does not signal an error if *program* exits with a non-zero exit status.

40.4 Creating an Asynchronous Process

In this section, we describe how to create an *asynchronous process*. After an asynchronous process is created, it runs in parallel with Emacs, and Emacs can communicate with it using the functions described in the following sections (see Section 40.7 [Input to Processes], page 1099, and see Section 40.9 [Output from Processes], page 1102). Note that process communication is only partially asynchronous: Emacs sends and receives data to and from a process only when those functions are called.

An asynchronous process is controlled either via a *pty* (pseudo-terminal) or a *pipe*. The choice of pty or pipe is made when creating the process, by default based on the value of the variable `process-connection-type` (see below). If available, ptys are usually preferable for processes visible to the user, as in Shell mode, because they allow for job control (`C-c`, `C-z`, etc.) between the process and its children, and because interactive programs treat ptys as terminal devices, whereas pipes don't support these features. However, for subprocesses used by Lisp programs for internal purposes (i.e., no user interaction with the subprocess is required), where significant amounts of data need to be exchanged between the subprocess and the Lisp program, it is often better to use a pipe, because pipes are more efficient. Also, the total number of ptys is limited on many systems, and it is good not to waste them unnecessarily.

`make-process` **&rest** *args* [Function]
: This function is the basic low-level primitive for starting asynchronous subprocesses. It returns a process object representing the subprocess. Compared to the more high-level `start-process`, described below, it takes keyword arguments, is more flexible, and allows to specify process filters and sentinels in a single call.

  The arguments *args* are a list of keyword/argument pairs. Omitting a keyword is always equivalent to specifying it with value `nil`. Here are the meaningful keywords:

  :name *name*
  : Use the string *name* as the process name; if a process with this name already exists, then *name* is modified (by appending '`<1>`', etc.) to be unique.

:buffer *buffer*
: Use *buffer* as the process buffer. If the value is `nil`, the subprocess is not associated with any buffer.

:command *command*
: Use *command* as the command line of the process. The value should be a list starting with the program's executable file name, followed by strings to give to the program as its arguments. If the first element of the list is `nil`, Emacs opens a new pseudoterminal (pty) and associates its input and output with *buffer*, without actually running any program; the rest of the list elements are ignored in that case.

:coding *coding*
: If *coding* is a symbol, it specifies the coding system to be used for both reading and writing of data from and to the connection. If *coding* is a cons cell `(`*`decoding`*` . `*`encoding`*`)`, then *decoding* will be used for reading and *encoding* for writing. The coding system used for encoding the data written to the program is also used for encoding the command-line arguments (but not the program itself, whose file name is encoded as any other file name; see Section 34.10.2 [Encoding and I/O], page 981).

  If *coding* is `nil`, the default rules for finding the coding system will apply. See Section 34.10.5 [Default Coding Systems], page 986.

:connection-type *type*
: Initialize the type of device used to communicate with the subprocess. Possible values are `pty` to use a pty, `pipe` to use a pipe, or `nil` to use the default derived from the value of the `process-connection-type` variable. If *type* is a cons cell `(`*`input`*` . `*`output`*`)`, then *input* will be used for standard input and *output* for standard output (and standard error if `:stderr` is `nil`).

  On systems where ptys are not available (MS-Windows), this parameter is ignored, and pipes are used unconditionally.

:noquery *query-flag*
: Initialize the process query flag to *query-flag*. See Section 40.11 [Query Before Exit], page 1110.

:stop *stopped*
: If provided, *stopped* must be `nil`; it is an error to use any non-`nil` value. The `:stop` key is ignored otherwise and is retained for compatibility with other process types such as pipe processes. Asynchronous subprocesses never start in the stopped state.

:filter *filter*
: Initialize the process filter to *filter*. If not specified, a default filter will be provided, which can be overridden later. See Section 40.9.2 [Filter Functions], page 1104.

:sentinel *sentinel*
: Initialize the process sentinel to *sentinel*. If not specified, a default sentinel will be used, which can be overridden later. See Section 40.10 [Sentinels], page 1109.

:stderr *stderr*
: Associate *stderr* with the standard error of the process. A non-`nil` value should be either a buffer or a pipe process created with `make-pipe-process`, described below. If *stderr* is `nil`, standard error is mixed with standard output, and both are sent to *buffer* or *filter*.

  If *stderr* is a buffer, Emacs will create a pipe process, the *standard error process*. This process will have the default filter (see Section 40.9.2 [Filter Functions], page 1104), sentinel (see Section 40.10 [Sentinels], page 1109), and coding systems (see Section 34.10.5 [Default Coding Systems], page 986). On the other hand, it will use *query-flag* as its query-on-exit flag (see Section 40.11 [Query Before Exit], page 1110). It will be associated with the *stderr* buffer (see Section 40.9.1 [Process Buffers], page 1102) and send its output (which is the standard error of the main process) there. To get the process object for the standard error process, pass the *stderr* buffer to `get-buffer-process`.

  If *stderr* is a pipe process, Emacs will use it as standard error process for the new process.

:file-handler *file-handler*
: If *file-handler* is non-`nil`, then look for a file name handler for the current buffer's `default-directory`, and invoke that file name handler to make the process. If there is no such handler, proceed as if *file-handler* were `nil`.

The original argument list, modified with the actual connection information, is available via the `process-contact` function.

The current working directory of the subprocess is set to the current buffer's value of `default-directory` if that is local (as determined by `unhandled-file-name-directory`), or `~` otherwise. If you want to run a process in a remote directory, pass `:file-handler t` to `make-process`. In that case, the current working directory is the local name component of `default-directory` (as determined by `file-local-name`).

Depending on the implementation of the file name handler, it might not be possible to apply *filter* or *sentinel* to the resulting process object. The `:stderr` argument cannot be a pipe process, file name handlers do not support pipe processes for this. A buffer as `:stderr` argument is accepted, its contents is shown without the use of pipe processes. See Section 40.9.2 [Filter Functions], page 1104, Section 40.10 [Sentinels], page 1109, and Section 40.9.4 [Accepting Output], page 1107.

Some file name handlers may not support `make-process`. In such cases, this function does nothing and returns `nil`.

`make-pipe-process` **&rest** *args* [Function]

This function creates a bidirectional pipe which can be attached to a child process. This is useful with the `:stderr` keyword of `make-process`. The function returns a process object.

The arguments *args* are a list of keyword/argument pairs. Omitting a keyword is always equivalent to specifying it with value `nil`.

Here are the meaningful keywords:

:name *name*
: Use the string *name* as the process name. As with `make-process`, it is modified if necessary to make it unique.

:buffer *buffer*
: Use *buffer* as the process buffer.

:coding *coding*
: If *coding* is a symbol, it specifies the coding system to be used for both reading and writing of data from and to the connection. If *coding* is a cons cell (*decoding* . *encoding*), then *decoding* will be used for reading and *encoding* for writing.

  If *coding* is `nil`, the default rules for finding the coding system will apply. See Section 34.10.5 [Default Coding Systems], page 986.

:noquery *query-flag*
: Initialize the process query flag to *query-flag*. See Section 40.11 [Query Before Exit], page 1110.

:stop *stopped*
: If *stopped* is non-`nil`, start the process in the stopped state. In the stopped state, a pipe process does not accept incoming data, but you can send outgoing data. The stopped state is set by `stop-process` and cleared by `continue-process` (see Section 40.8 [Signals to Processes], page 1100).

:filter *filter*
: Initialize the process filter to *filter*. If not specified, a default filter will be provided, which can be changed later. See Section 40.9.2 [Filter Functions], page 1104.

:sentinel *sentinel*
: Initialize the process sentinel to *sentinel*. If not specified, a default sentinel will be used, which can be changed later. See Section 40.10 [Sentinels], page 1109.

The original argument list, modified with the actual connection information, is available via the `process-contact` function.

`start-process` *name buffer-or-name program* **&rest** *args* [Function]

This function is a higher-level wrapper around `make-process`, exposing an interface that is similar to `call-process`. It creates a new asynchronous subprocess and starts the specified *program* running in it. It returns a process object that stands for the new

subprocess in Lisp. The argument *name* specifies the name for the process object; as with `make-process`, it is modified if necessary to make it unique. The buffer *buffer-or-name* is the buffer to associate with the process.

If *program* is `nil`, Emacs opens a new pseudoterminal (pty) and associates its input and output with *buffer-or-name*, without creating a subprocess. In that case, the remaining arguments *args* are ignored.

The rest of *args* are strings that specify command line arguments for the subprocess.

In the example below, the first process is started and runs (rather, sleeps) for 100 seconds (the output buffer '`foo`' is created immediately). Meanwhile, the second process is started, and given the name '`my-process<1>`' for the sake of uniqueness. It inserts the directory listing at the end of the buffer '`foo`', before the first process finishes. Then it finishes, and a message to that effect is inserted in the buffer. Much later, the first process finishes, and another message is inserted in the buffer for it.

```
(start-process "my-process" "foo" "sleep" "100")
     ⇒ #<process my-process>

(start-process "my-process" "foo" "ls" "-l" "/bin")
     ⇒ #<process my-process<1>>

---------- Buffer: foo ----------
total 8336
-rwxr-xr-x 1 root root 971384 Mar 30 10:14 bash
-rwxr-xr-x 1 root root 146920 Jul  5  2011 bsd-csh
...
-rwxr-xr-x 1 root root 696880 Feb 28 15:55 zsh4

Process my-process<1> finished

Process my-process finished
---------- Buffer: foo ----------
```

`start-file-process` *name buffer-or-name program* **&rest** *args* [Function]

Like `start-process`, this function starts a new asynchronous subprocess running *program* in it, and returns its process object.

The difference from `start-process` is that this function may invoke a file name handler based on the value of `default-directory`. This handler ought to run *program*, perhaps on the local host, perhaps on a remote host that corresponds to `default-directory`. In the latter case, the local part of `default-directory` becomes the working directory of the process.

This function does not try to invoke file name handlers for *program* or for the rest of *args*. For that reason, if *program* or any of *args* use the remote-file syntax (see Section 26.12 [Magic File Names], page 651), they must be converted either to file names relative to `default-directory`, or to names that identify the files locally on the remote host, by running them through `file-local-name`.

Depending on the implementation of the file name handler, it might not be possible to apply `process-filter` or `process-sentinel` to the resulting process object. See Section 40.9.2 [Filter Functions], page 1104, and Section 40.10 [Sentinels], page 1109.

Some file name handlers may not support `start-file-process` (for example the function `ange-ftp-hook-function`). In such cases, this function does nothing and returns `nil`.

`start-process-shell-command` *name buffer-or-name command* [Function]
This function is like `start-process`, except that it uses a shell to execute the specified *command*. The argument *command* is a shell command string. The variable `shell-file-name` specifies which shell to use.

The point of running a program through the shell, rather than directly with `make-process` or `start-process`, is so that you can employ shell features such as wildcards in the arguments. It follows that if you include any arbitrary user-specified arguments in the command, you should quote them with `shell-quote-argument` first, so that any special shell characters do *not* have their special shell meanings. See Section 40.2 [Shell Arguments], page 1083. Of course, when executing commands based on user input you should also consider the security implications.

`start-file-process-shell-command` *name buffer-or-name command* [Function]
This function is like `start-process-shell-command`, but uses `start-file-process` internally. Because of this, *command* can also be executed on remote hosts, depending on `default-directory`.

`process-connection-type` [Variable]
This variable controls the type of device used to communicate with asynchronous subprocesses. If it is non-`nil`, then ptys are used, when available. Otherwise, pipes are used.

The value of `process-connection-type` takes effect when `make-process` or `start-process` is called. So you can specify how to communicate with one subprocess by binding the variable around the call to these functions.

Note that the value of this variable is ignored when `make-process` is called with a non-`nil` value of the `:stderr` parameter; in that case, Emacs will communicate with the process using pipes. It is also ignored if ptys are unavailable (MS-Windows).

```
(let ((process-connection-type nil))  ; use a pipe
  (start-process ...))
```

To determine whether a given subprocess actually got a pipe or a pty, use the function `process-tty-name` (see Section 40.6 [Process Information], page 1095).

`process-error-pause-time` [Variable]
If a process sentinel/filter function has an error, Emacs will (by default) pause Emacs for `process-error-pause-time` seconds after displaying this error, so that users will see the error in question. However, this can lead to situations where Emacs becomes unresponsive (if there's a lot of these errors happening), so this can be disabled by setting `process-error-pause-time` to 0.

40.5 Deleting Processes

Deleting a process disconnects Emacs immediately from the subprocess. Processes are deleted automatically after they terminate, but not necessarily right away. You can delete

a process explicitly at any time. If you explicitly delete a terminated process before it is deleted automatically, no harm results. Deleting a running process sends a signal to terminate it (and its child processes, if any), and calls the process sentinel. See Section 40.10 [Sentinels], page 1109.

When a process is deleted, the process object itself continues to exist as long as other Lisp objects point to it. All the Lisp primitives that work on process objects accept deleted processes, but those that do I/O or send signals will report an error. The process mark continues to point to the same place as before, usually into a buffer where output from the process was being inserted.

`delete-exited-processes` [User Option]

This variable controls automatic deletion of processes that have terminated (due to calling `exit` or to a signal). If it is `nil`, then they continue to exist until the user runs `list-processes`. Otherwise, they are deleted immediately after they exit.

`delete-process` **&optional** *process* [Function]

This function deletes a process, killing it with a `SIGKILL` signal if the process was running a program. The argument may be a process, the name of a process, a buffer, or the name of a buffer. (A buffer or buffer-name stands for the process that `get-buffer-process` returns, and a missing or `nil` *process* means that the current buffer's process should be killed.) Calling `delete-process` on a running process terminates it, updates the process status, and runs the sentinel immediately. If the process has already terminated, calling `delete-process` has no effect on its status, or on the running of its sentinel (which will happen sooner or later).

If the process object represents a network, serial, or pipe connection, its status changes to `closed`; otherwise, it changes to `signal`, unless the process already exited. See Section 40.6 [Process Information], page 1095.

```
(delete-process "*shell*")
     ⇒ nil
```

40.6 Process Information

Several functions return information about processes.

`list-processes` **&optional** *query-only buffer* [Command]

This command displays a listing of all living processes. In addition, it finally deletes any process whose status was '`Exited`' or '`Signaled`'. It returns `nil`.

The processes are shown in a buffer named `*Process List*` (unless you specify otherwise using the optional argument *buffer*), whose major mode is Process Menu mode.

If *query-only* is non-`nil`, it only lists processes whose query flag is non-`nil`. See Section 40.11 [Query Before Exit], page 1110.

`process-list` [Function]

This function returns a list of all processes that have not been deleted.

```
(process-list)
     ⇒ (#<process display-time> #<process shell>)
```

`num-processors` **&optional** *query* [Function]

This function returns the number of processors, a positive integer. Each usable thread execution unit counts as a processor. By default, the count includes the number of available processors, which you can override by setting the `OMP_NUM_THREADS` environment variable of OpenMP (`https://www.openmp.org/spec-html/5.1/openmpse59.html`). If the optional argument *query* is `current`, this function ignores `OMP_NUM_THREADS`; if *query* is `all`, this function also counts processors that are on the system but are not available to the current process.

`get-process` *name* [Function]

This function returns the process named *name* (a string), or `nil` if there is none. The argument *name* can also be a process object, in which case it is returned.

```
(get-process "shell")
     ⇒ #<process shell>
```

`process-command` *process* [Function]

This function returns the command that was executed to start *process*. This is a list of strings, the first string being the program executed and the rest of the strings being the arguments that were given to the program. For a network, serial, or pipe connection, this is either `nil`, which means the process is running or `t` (process is stopped).

```
(process-command (get-process "shell"))
     ⇒ ("bash" "-i")
```

`process-contact` *process* **&optional** *key no-block* [Function]

This function returns information about how a network, a serial, or a pipe connection was set up. When *key* is `nil`, it returns `(`*hostname service*`)` for a network connection, `(`*port speed*`)` for a serial connection, and `t` for a pipe connection. For an ordinary child process, this function always returns `t` when called with a `nil` *key*.

If *key* is `t`, the value is the complete status information for the connection, server, serial port, or pipe; that is, the list of keywords and values specified in `make-network-process`, `make-serial-process`, or `make-pipe-process`, except that some of the values represent the current status instead of what you specified.

For a network process, the values include (see `make-network-process` for a complete list):

`:buffer` The associated value is the process buffer.

`:filter` The associated value is the process filter function. See Section 40.9.2 [Filter Functions], page 1104.

`:sentinel` The associated value is the process sentinel function. See Section 40.10 [Sentinels], page 1109.

`:remote` In a connection, the address in internal format of the remote peer.

`:local` The local address, in internal format.

`:service` In a server, if you specified `t` for *service*, this value is the actual port number.

`:local` and `:remote` are included even if they were not specified explicitly in `make-network-process`.

For a serial connection, see `make-serial-process` and `serial-process-configure` for the list of keys. For a pipe connection, see `make-pipe-process` for the list of keys.

If *key* is a keyword, the function returns the value corresponding to that keyword.

If *process* is a non-blocking network stream that hasn't been fully set up yet, then this function will block until that has happened. If given the optional *no-block* parameter, this function will return `nil` instead of blocking.

`process-id` *process* [Function]

This function returns the PID of *process*. This is an integral number that distinguishes the process *process* from all other processes running on the same computer at the current time. The PID of a process is chosen by the operating system kernel when the process is started and remains constant as long as the process exists. For network, serial, and pipe connections, this function returns `nil`.

`process-name` *process* [Function]

This function returns the name of *process*, as a string.

`process-status` *process-name* [Function]

This function returns the status of *process-name* as a symbol. The argument *process-name* must be a process, a buffer, or a process name (a string).

The possible values for an actual subprocess are:

`run` for a process that is running.

`stop` for a process that is stopped but continuable.

`exit` for a process that has exited.

`signal` for a process that has received a fatal signal.

`open` for a network, serial, or pipe connection that is open.

`closed` for a network, serial, or pipe connection that is closed. Once a connection is closed, you cannot reopen it, though you might be able to open a new connection to the same place.

`connect` for a non-blocking connection that is waiting to complete.

`failed` for a non-blocking connection that has failed to complete.

`listen` for a network server that is listening.

`nil` if *process-name* is not the name of an existing process.

```
(process-status (get-buffer "*shell*"))
     ⇒ run
```

For a network, serial, or pipe connection, `process-status` returns one of the symbols `open`, `stop`, or `closed`. The latter means that the other side closed the connection, or Emacs did `delete-process`. The value `stop` means that `stop-process` was called on the connection.

`process-live-p` *process* [Function]
: This function returns non-`nil` if *process* is alive. A process is considered alive if its status is `run`, `open`, `listen`, `connect` or `stop`.

`process-type` *process* [Function]
: This function returns the symbol `network` for a network connection or server, `serial` for a serial port connection, `pipe` for a pipe connection, or `real` for a subprocess created for running a program.

`process-exit-status` *process* [Function]
: This function returns the exit status of *process* or the signal number that killed it. (Use the result of `process-status` to determine which of those it is.) If *process* has not yet terminated, the value is 0. For network, serial, and pipe connections that are already closed, the value is either 0 or 256, depending on whether the connection was closed normally or abnormally.

`process-tty-name` *process* **&optional** *stream* [Function]
: This function returns the terminal name that *process* is using for its communication with Emacs—or `nil` if it is using pipes instead of a pty (see `process-connection-type` in Section 40.4 [Asynchronous Processes], page 1089). By default, this function returns the terminal name if any of *process*'s standard streams use a terminal. If *stream* is one of `stdin`, `stdout`, or `stderr`, this function returns the terminal name (or `nil`, as above) that *process* uses for that stream specifically. You can use this to determine whether a particular stream uses a pipe or a pty.

  If *process* represents a program running on a remote host, this function returns the *local* terminal name that communicates with *process*; you can get the terminal name used by that program on the remote host with the process property `remote-tty`. If *process* represents a network, serial, or pipe connection, this function always returns `nil`.

`process-coding-system` *process* [Function]
: This function returns a cons cell `(`*`decode`*` . `*`encode`*`)`, describing the coding systems in use for decoding output from, and encoding input to, *process* (see Section 34.10 [Coding Systems], page 980).

`set-process-coding-system` *process* **&optional** *decoding-system* *encoding-system* [Function]
: This function specifies the coding systems to use for subsequent output from and input to *process*. It will use *decoding-system* to decode subprocess output, and *encoding-system* to encode subprocess input.

Every process also has a property list that you can use to store miscellaneous values associated with the process.

`process-get` *process propname* [Function]
: This function returns the value of the *propname* property of *process*.

`process-put` *process propname value* [Function]
: This function sets the value of the *propname* property of *process* to *value*.

`process-plist` *process* [Function]
This function returns the process plist of *process*.

`set-process-plist` *process plist* [Function]
This function sets the process plist of *process* to *plist*.

40.7 Sending Input to Processes

Asynchronous subprocesses receive input when it is sent to them by Emacs, which is done with the functions in this section. You must specify the process to send input to, and the input data to send. If the subprocess runs a program, the data appears on the standard input of that program; for connections, the data is sent to the connected device or program.

Some operating systems have limited space for buffered input in a pty. On these systems, Emacs sends an EOF periodically amidst the other characters, to force them through. For most programs, these EOFs do no harm.

Subprocess input is normally encoded using a coding system before the subprocess receives it, much like text written into a file. You can use `set-process-coding-system` to specify which coding system to use (see Section 40.6 [Process Information], page 1095). Otherwise, the coding system comes from `coding-system-for-write`, if that is non-`nil`; or else from the defaulting mechanism (see Section 34.10.5 [Default Coding Systems], page 986).

Sometimes the system is unable to accept input for that process, because the input buffer is full. When this happens, the send functions wait a short while, accepting output from subprocesses, and then try again. This gives the subprocess a chance to read more of its pending input and make space in the buffer. It also allows filters (including the one currently running), sentinels and timers to run—so take account of that in writing your code.

In these functions, the *process* argument can be a process or the name of a process, or a buffer or buffer name (which stands for a process via `get-buffer-process`). `nil` means the current buffer's process.

`process-send-string` *process string* [Function]
This function sends *process* the contents of *string* as standard input. It returns `nil`. For example, to make a Shell buffer list files:

```
(process-send-string "shell<1>" "ls\n")
     ⇒ nil
```

`process-send-region` *process start end* [Function]
This function sends the text in the region defined by *start* and *end* as standard input to *process*.

An error is signaled unless both *start* and *end* are integers or markers that indicate positions in the current buffer. (It is unimportant which number is larger.)

`process-send-eof` **&optional** *process* [Function]
This function makes *process* see an end-of-file in its input. The EOF comes after any text already sent to it. The function returns *process*.

```
(process-send-eof "shell")
     ⇒ "shell"
```

`process-running-child-p` **&optional** *process* [Function]

This function will tell you whether a *process*, which must not be a connection but a real subprocess, has given control of its terminal to a child process of its own. If this is true, the function returns the numeric ID of the foreground process group of *process*; it returns `nil` if Emacs can be certain that this is not so. The value is `t` if Emacs cannot tell whether this is true. This function signals an error if *process* is a network, serial, or pipe connection, or if the subprocess is not active.

40.8 Sending Signals to Processes

Sending a signal to a subprocess is a way of interrupting its activities. There are several different signals, each with its own meaning. The set of signals and their names is defined by the operating system. For example, the signal `SIGINT` means that the user has typed `C-c`, or that some analogous thing has happened.

Each signal has a standard effect on the subprocess. Most signals kill the subprocess, but some stop (or resume) execution instead. Most signals can optionally be handled by programs; if the program handles the signal, then we can say nothing in general about its effects.

You can send signals explicitly by calling the functions in this section. Emacs also sends signals automatically at certain times: killing a buffer sends a `SIGHUP` signal to all its associated processes; killing Emacs sends a `SIGHUP` signal to all remaining processes. (`SIGHUP` is a signal that usually indicates that the user "hung up the phone", i.e., disconnected.)

Each of the signal-sending functions takes two optional arguments: *process* and *current-group*.

The argument *process* must be either a process, a process name, a buffer, a buffer name, or `nil`. A buffer or buffer name stands for a process through `get-buffer-process`. `nil` stands for the process associated with the current buffer. Except with `stop-process` and `continue-process`, an error is signaled if *process* does not identify an active process, or if it represents a network, serial, or pipe connection.

The argument *current-group* is a flag that makes a difference when you are running a job-control shell as an Emacs subprocess. If it is non-`nil`, then the signal is sent to the current process-group of the terminal that Emacs uses to communicate with the subprocess. If the process is a job-control shell, this means the shell's current subjob. If *current-group* is `nil`, the signal is sent to the process group of the immediate subprocess of Emacs. If the subprocess is a job-control shell, this is the shell itself. If *current-group* is `lambda`, the signal is sent to the process-group that owns the terminal, but only if it is not the shell itself.

The flag *current-group* has no effect when a pipe is used to communicate with the subprocess, because the operating system does not support the distinction in the case of pipes. For the same reason, job-control shells won't work when a pipe is used. See `process-connection-type` in Section 40.4 [Asynchronous Processes], page 1089.

`interrupt-process` **&optional** *process current-group* [Function]

This function interrupts the process *process* by sending the signal `SIGINT`. Outside of Emacs, typing the interrupt character (normally `C-c` on some systems, and `DEL` on others) sends this signal. When the argument *current-group* is non-`nil`, you can

think of this function as typing `C-c` on the terminal by which Emacs talks to the subprocess.

`kill-process` **&optional** *process current-group* [Command]
This command kills the process *process* by sending the signal `SIGKILL`. This signal kills the subprocess immediately, and cannot be handled by the subprocess. Interactively, it'll prompt the user for a process name, defaulting to the process (if any) in the current buffer.

`quit-process` **&optional** *process current-group* [Function]
This function sends the signal `SIGQUIT` to the process *process*. This signal is the one sent by the quit character (usually `C-\`) when you are not inside Emacs.

`stop-process` **&optional** *process current-group* [Function]
This function stops the specified *process*. If it is a real subprocess running a program, it sends the signal `SIGTSTP` to that subprocess. If *process* represents a network, serial, or pipe connection, this function inhibits handling of the incoming data from the connection; for a network server, this means not accepting new connections. Use `continue-process` to resume normal execution.

Outside of Emacs, on systems with job control, the stop character (usually `C-z`) normally sends the `SIGTSTP` signal to a subprocess. When *current-group* is non-`nil`, you can think of this function as typing `C-z` on the terminal Emacs uses to communicate with the subprocess.

`continue-process` **&optional** *process current-group* [Function]
This function resumes execution of the process *process*. If it is a real subprocess running a program, it sends the signal `SIGCONT` to that subprocess; this presumes that *process* was stopped previously. If *process* represents a network, serial, or pipe connection, this function resumes handling of the incoming data from the connection. For serial connections, data that arrived during the time the process was stopped might be lost.

`signal-process` *process signal* **&optional** *remote* [Command]
This function sends a signal to process *process*. The argument *signal* specifies which signal to send; it should be an integer, or a symbol whose name is a signal.

The *process* argument can be a system process ID (an integer); that allows you to send signals to processes that are not children of Emacs. See Section 40.12 [System Processes], page 1111.

If *process* is a process object which contains the property `remote-pid`, or *process* is a number and *remote* is a remote file name, *process* is interpreted as process on the respective remote host, which will be the process to signal.

If *process* is a string, it is interpreted as process object with the respective process name, or as a number.

Sometimes, it is necessary to send a signal to a non-local asynchronous process. This is possible by writing an own `interrupt-process` or `signal-process` implementation. This function must be added then to `interrupt-process-functions` or `signal-process-functions`, respectively.

`interrupt-process-functions` [Variable]

This variable is a list of functions to be called for `interrupt-process`. The arguments of the functions are the same as for `interrupt-process`. These functions are called in the order of the list, until one of them returns non-`nil`. The default function, which shall always be the last in this list, is `internal-default-interrupt-process`.

This is the mechanism, how Tramp implements `interrupt-process`.

`signal-process-functions` [Variable]

This variable is a list of functions to be called for `signal-process`. The arguments of the functions are the same as for `signal-process`. These functions are called in the order of the list, until one of them returns non-`nil`. The default function, which shall always be the last in this list, is `internal-default-signal-process`.

This is the mechanism, how Tramp implements `signal-process`.

40.9 Receiving Output from Processes

The output that an asynchronous subprocess writes to its standard output stream is passed to a function called the *filter function*. The default filter function simply inserts the output into a buffer, which is called the associated buffer of the process (see Section 40.9.1 [Process Buffers], page 1102). If the process has no buffer then the default filter discards the output.

If the subprocess writes to its standard error stream, by default the error output is also passed to the process filter function. Alternatively, you could use the `:stderr` parameter with a non-`nil` value in a call to `make-process` (see Section 40.4 [Asynchronous Processes], page 1089) to make the destination of the error output separate from the standard output.

When a subprocess terminates, Emacs reads any pending output, then stops reading output from that subprocess. Therefore, if the subprocess has children that are still live and still producing output, Emacs won't receive that output.

Output from a subprocess can arrive only while Emacs is waiting: when reading terminal input (see the function `waiting-for-user-input-p`), in `sit-for` and `sleep-for` (see Section 22.10 [Waiting], page 471), in `accept-process-output` (see Section 40.9.4 [Accepting Output], page 1107), and in functions which send data to processes (see Section 40.7 [Input to Processes], page 1099). This minimizes the problem of timing errors that usually plague parallel programming. For example, you can safely create a process and only then specify its buffer or filter function; no output can arrive before you finish, if the code in between does not call any primitive that waits.

`process-adaptive-read-buffering` [Variable]

On some systems, when Emacs reads the output from a subprocess, the output data is read in very small blocks, potentially resulting in very poor performance. This behavior can be remedied to some extent by setting the variable `process-adaptive-read-buffering` to a non-`nil` value (the default), as it will automatically delay reading from such processes, thus allowing them to produce more output before Emacs tries to read it.

40.9.1 Process Buffers

A process can (and usually does) have an *associated buffer*, which is an ordinary Emacs buffer that is used for two purposes: storing the output from the process, and deciding when

to kill the process. You can also use the buffer to identify a process to operate on, since in normal practice only one process is associated with any given buffer. Many applications of processes also use the buffer for editing input to be sent to the process, but this is not built into Emacs Lisp.

By default, process output is inserted in the associated buffer. (You can change this by defining a custom filter function, see Section 40.9.2 [Filter Functions], page 1104.) The position to insert the output is determined by the `process-mark`, which is then updated to point to the end of the text just inserted. Usually, but not always, the `process-mark` is at the end of the buffer.

Killing the associated buffer of a process also kills the process. Emacs asks for confirmation first, if the process's `process-query-on-exit-flag` is non-`nil` (see Section 40.11 [Query Before Exit], page 1110). This confirmation is done by the function `process-kill-buffer-query-function`, which is run from `kill-buffer-query-functions` (see Section 28.10 [Killing Buffers], page 687).

`process-buffer` *process* [Function]

This function returns the associated buffer of the specified *process*.

```
(process-buffer (get-process "shell"))
     ⇒ #<buffer *shell*>
```

`process-mark` *process* [Function]

This function returns the process marker for *process*, which is the marker that says where to insert output from the process.

If *process* does not have a buffer, `process-mark` returns a marker that points nowhere.

The default filter function uses this marker to decide where to insert process output, and updates it to point after the inserted text. That is why successive batches of output are inserted consecutively.

Custom filter functions normally should use this marker in the same fashion. For an example of a filter function that uses `process-mark`, see [Process Filter Example], page 1105.

When the user is expected to enter input in the process buffer for transmission to the process, the process marker separates the new input from previous output.

`set-process-buffer` *process buffer* [Function]

This function sets the buffer associated with *process* to *buffer*. If *buffer* is `nil`, the process becomes associated with no buffer; if non-`nil` and different from the buffer associated with the process, the process mark will be set to point to the end of *buffer* (unless the process mark is already associated with *buffer*).

`get-buffer-process` *buffer-or-name* [Function]

This function returns a nondeleted process associated with the buffer specified by *buffer-or-name*. If there are several processes associated with it, this function chooses one (currently, the one most recently created, but don't count on that). Deletion of a process (see `delete-process`) makes it ineligible for this function to return.

It is usually a bad idea to have more than one process associated with the same buffer.

```
(get-buffer-process "*shell*")
     ⇒ #<process shell>
```

Killing the process's buffer deletes the process, which kills the subprocess with a `SIGHUP` signal (see Section 40.8 [Signals to Processes], page 1100).

If the process's buffer is displayed in a window, your Lisp program may wish to tell the process the dimensions of that window, so that the process could adapt its output to those dimensions, much as it adapts to the screen dimensions. The following functions allow communicating this kind of information to processes; however, not all systems support the underlying functionality, so it is best to provide fallbacks, e.g., via command-line arguments or environment variables.

`set-process-window-size` *process height width* [Function]
Tell *process* that its logical window size has dimensions *width* by *height*, in character units. If this function succeeds in communicating this information to the process, it returns `t`; otherwise it returns `nil`.

When windows that display buffers associated with process change their dimensions, the affected processes should be told about these changes. By default, when the window configuration changes, Emacs will automatically call `set-process-window-size` on behalf of every process whose buffer is displayed in a window, passing it the smallest dimensions of all the windows displaying the process's buffer. This works via `window-configuration-change-hook` (see Section 29.28 [Window Hooks], page 782), which is told to invoke the function that is the value of the variable `window-adjust-process-window-size-function` for each process whose buffer is displayed in at least one window. You can customize this behavior by setting the value of that variable.

`window-adjust-process-window-size-function` [User Option]
The value of this variable should be a function of two arguments: a process and the list of windows displaying the process's buffer. When the function is called, the process's buffer is the current buffer. The function should return a cons cell `(`*`width`*` . `*`height`*`)` that describes the dimensions of the logical process window to be passed via a call to `set-process-window-size`. The function can also return `nil`, in which case Emacs will not call `set-process-window-size` for this process.

Emacs supplies two predefined values for this variable: `window-adjust-process-window-size-smallest`, which returns the smallest of all the dimensions of the windows that display a process's buffer; and `window-adjust-process-window-size-largest`, which returns the largest dimensions. For more complex strategies, write your own function.

This variable can be buffer-local.

If the process has the `adjust-window-size-function` property (see Section 40.6 [Process Information], page 1095), its value overrides the global and buffer-local values of `window-adjust-process-window-size-function`.

40.9.2 Process Filter Functions

A process *filter function* is a function that receives the standard output from the associated process. *All* output from that process is passed to the filter. The default filter simply outputs directly to the process buffer.

By default, the error output from the process, if any, is also passed to the filter function, unless the destination for the standard error stream of the process was separated from the standard output when the process was created. Emacs will only call the filter function during certain function calls. See Section 40.9 [Output from Processes], page 1102. Note that if any of those functions are called by the filter, the filter may be called recursively.

A filter function must accept two arguments: the associated process and a string, which is output just received from it. The function is then free to do whatever it chooses with the output.

Quitting is normally inhibited within a filter function—otherwise, the effect of typing `C-g` at command level or to quit a user command would be unpredictable. If you want to permit quitting inside a filter function, bind `inhibit-quit` to `nil`. In most cases, the right way to do this is with the macro `with-local-quit`. See Section 22.11 [Quitting], page 472.

If an error happens during execution of a filter function, it is caught automatically, so that it doesn't stop the execution of whatever program was running when the filter function was started. However, if `debug-on-error` is non-`nil`, errors are not caught. This makes it possible to use the Lisp debugger to debug filter functions. See Section 19.1 [Debugger], page 334. If an error is caught, Emacs pauses for `process-error-pause-time` seconds so that the user sees the error. See Section 40.4 [Asynchronous Processes], page 1089.

Many filter functions sometimes (or always) insert the output in the process's buffer, mimicking the actions of the default filter. Such filter functions need to make sure that they save the current buffer, select the correct buffer (if different) before inserting output, and then restore the original buffer. They should also check whether the buffer is still alive, update the process marker, and in some cases update the value of point. Here is how to do these things:

```
(defun ordinary-insertion-filter (proc string)
  (when (buffer-live-p (process-buffer proc))
    (with-current-buffer (process-buffer proc)
      (let ((moving (= (point) (process-mark proc))))
        (save-excursion
          ;; Insert the text, advancing the process marker.
          (goto-char (process-mark proc))
          (insert string)
          (set-marker (process-mark proc) (point)))
        (if moving (goto-char (process-mark proc)))))))
```

To make the filter force the process buffer to be visible whenever new text arrives, you could insert a line like the following just before the `with-current-buffer` construct:

```
(display-buffer (process-buffer proc))
```

To force point to the end of the new output, no matter where it was previously, eliminate the variable `moving` from the example and call `goto-char` unconditionally. Note that this doesn't necessarily move the window point. The default filter actually uses `insert-before-markers` which moves all markers, including the window point. This may move unrelated markers, so it's generally better to move the window point explicitly, or set its insertion type to `t` (see Section 29.19 [Window Point], page 761).

Note that Emacs automatically saves and restores the match data while executing filter functions. See Section 35.6 [Match Data], page 1014.

The output to the filter may come in chunks of any size. A program that produces the same output twice in a row may send it as one batch of 200 characters one time, and five

batches of 40 characters the next. If the filter looks for certain text strings in the subprocess output, make sure to handle the case where one of these strings is split across two or more batches of output; one way to do this is to insert the received text into a temporary buffer, which can then be searched.

`set-process-filter` *process filter* [Function]
: This function gives *process* the filter function *filter*. If *filter* is `nil`, it gives the process the default filter, which inserts the process output into the process buffer. If *filter* is `t`, Emacs stops accepting output from the process, unless it's a network server process that listens for incoming connections.

`process-filter` *process* [Function]
: This function returns the filter function of *process*.

In case the process's output needs to be passed to several filters, you can use `add-function` to combine an existing filter with a new one. See Section 13.12 [Advising Functions], page 252.

Here is an example of the use of a filter function:

```
(defun keep-output (process output)
   (setq kept (cons output kept)))
     ⇒ keep-output
(setq kept nil)
     ⇒ nil
(set-process-filter (get-process "shell") 'keep-output)
     ⇒ keep-output
(process-send-string "shell" "ls ~/other\n")
     ⇒ nil
kept
     ⇒ ("lewis@slug:$ "
"FINAL-W87-SHORT.MSS    backup.otl              kolstad.mss~
address.txt             backup.psf              kolstad.psf
backup.bib~             david.mss               resume-Dec-86.mss~
backup.err              david.psf               resume-Dec.psf
backup.mss              dland                   syllabus.mss
"
"#backups.mss#          backup.mss~             kolstad.mss
")
```

40.9.3 Decoding Process Output

When Emacs writes process output directly into a multibyte buffer, it decodes the output according to the process output coding system. If the coding system is `raw-text` or `no-conversion`, Emacs converts the unibyte output to multibyte using `string-to-multibyte`, and inserts the resulting multibyte text.

You can use `set-process-coding-system` to specify which coding system to use (see Section 40.6 [Process Information], page 1095). Otherwise, the coding system comes from `coding-system-for-read`, if that is non-`nil`; or else from the defaulting mechanism (see Section 34.10.5 [Default Coding Systems], page 986). If the text output by a process contains null bytes, Emacs by default uses `no-conversion` for it; see Section 34.10.3 [Lisp and Coding Systems], page 982, for how to control this behavior.

Warning: Coding systems such as `undecided`, which determine the coding system from the data, do not work entirely reliably with asynchronous subprocess output. This is because

Emacs has to process asynchronous subprocess output in batches, as it arrives. Emacs must try to detect the proper coding system from one batch at a time, and this does not always work. Therefore, if at all possible, specify a coding system that determines both the character code conversion and the end of line conversion—that is, one like `latin-1-unix`, rather than `undecided` or `latin-1`.

When Emacs calls a process filter function, it provides the process output as a multibyte string or as a unibyte string according to the process's filter coding system. Emacs decodes the output according to the process output coding system, which usually produces a multibyte string, except for coding systems such as `binary` and `raw-text`.

40.9.4 Accepting Output from Processes

Output from asynchronous subprocesses normally arrives only while Emacs is waiting for some sort of external event, such as elapsed time or terminal input. Occasionally it is useful in a Lisp program to explicitly permit output to arrive at a specific point, or even to wait until output arrives from a process.

`accept-process-output` **&optional** *process seconds millisec just-this-one* [Function]

This function allows Emacs to read pending output from processes. The output is given to their filter functions. If *process* is non-`nil` then this function does not return until some output has been received from *process* or *process* has closed the connection.

The arguments *seconds* and *millisec* let you specify timeout periods. The former specifies a period measured in seconds and the latter specifies one measured in milliseconds. The two time periods thus specified are added together, and `accept-process-output` returns after that much time, even if there is no subprocess output.

The argument *millisec* is obsolete (and should not be used), because *seconds* can be floating point to specify waiting a fractional number of seconds. If *seconds* is 0, the function accepts whatever output is pending but does not wait.

If *process* is a process, and the argument *just-this-one* is non-`nil`, only output from that process is handled, suspending output from other processes until some output has been received from that process or the timeout expires. If *just-this-one* is an integer, also inhibit running timers. This feature is generally not recommended, but may be necessary for specific applications, such as speech synthesis.

The function `accept-process-output` returns non-`nil` if it got output from *process*, or from any process if *process* is `nil`; this can occur even after a process has exited if the corresponding connection contains buffered data. The function returns `nil` if the timeout expired or the connection was closed before output arrived.

If a connection from a process contains buffered data, `accept-process-output` can return non-`nil` even after the process has exited. Therefore, although the following loop:

```
;; This loop contains a bug.
(while (process-live-p process)
  (accept-process-output process))
```

will often read all output from *process*, it has a race condition and can miss some output if `process-live-p` returns `nil` while the connection still contains data. Better is to write the loop like this:

```
(while (accept-process-output process))
```

If you have passed a non-`nil` *stderr* to `make-process`, it will have a standard error process. See Section 40.4 [Asynchronous Processes], page 1089. In that case, waiting for process output from the main process doesn't wait for output from the standard error process. To make sure you have received both all of standard output and all of standard error from a process, use the following code:

```
(while (accept-process-output process))
(while (accept-process-output stderr-process))
```

If you passed a buffer to the *stderr* argument of `make-process`, you still have to wait for the standard error process, like so:

```
(let* ((stdout (generate-new-buffer "stdout"))
       (stderr (generate-new-buffer "stderr"))
       (process (make-process :name "test"
                              :command '("my-program")
                              :buffer stdout
                              :stderr stderr))
       (stderr-process (get-buffer-process stderr)))
  (unless (and process stderr-process)
    (error "Process unexpectedly nil"))
  (while (accept-process-output process))
  (while (accept-process-output stderr-process)))
```

Only when both `accept-process-output` forms return `nil`, you can be sure that the process has exited and Emacs has read all its output.

Reading pending standard error from a process running on a remote host is not possible this way.

40.9.5 Processes and Threads

Because threads were a relatively late addition to Emacs Lisp, and due to the way dynamic binding was sometimes used in conjunction with `accept-process-output`, by default a process is locked to the thread that created it. When a process is locked to a thread, output from the process can only be accepted by that thread.

A Lisp program can specify to which thread a process is to be locked, or instruct Emacs to unlock a process, in which case its output can be processed by any thread. Only a single thread will wait for output from a given process at one time—once one thread begins waiting for output, the process is temporarily locked until `accept-process-output` or `sit-for` returns.

If the thread exits, all the processes locked to it are unlocked.

`process-thread` *process* [Function]
: Return the thread to which *process* is locked. If *process* is unlocked, return `nil`.

`set-process-thread` *process thread* [Function]
: Set the locking thread of *process* to *thread*. *thread* may be `nil`, in which case the process is unlocked.

40.10 Sentinels: Detecting Process Status Changes

A *process sentinel* is a function that is called whenever the associated process changes status for any reason, including signals (whether sent by Emacs or caused by the process's own actions) that terminate, stop, or continue the process. The process sentinel is also called if the process exits. The sentinel receives two arguments: the process for which the event occurred, and a string describing the type of event.

If no sentinel function was specified for a process, it will use the default sentinel function, which inserts a message in the process's buffer with the process name and the string describing the event.

The string describing the event looks like one of the following (but this is not an exhaustive list of event strings):

- `"finished\n"`.
- `"deleted\n"`.
- `"exited abnormally with code` *`exitcode`* `(core dumped)\n"`. The "core dumped" part is optional, and only appears if the process dumped core.
- `"failed with code` *`fail-code`*`\n"`.
- `"`*`signal-description`* `(core dumped)\n"`. The *signal-description* is a system-dependent textual description of a signal, e.g., `"killed"` for `SIGKILL`. The "core dumped" part is optional, and only appears if the process dumped core.
- `"open from` *`host-name`*`\n"`.
- `"open\n"`.
- `"run\n"`.
- `"connection broken by remote peer\n"`.

A sentinel runs only while Emacs is waiting (e.g., for terminal input, or for time to elapse, or for process output). This avoids the timing errors that could result from running sentinels at random places in the middle of other Lisp programs. A program can wait, so that sentinels will run, by calling `sit-for` or `sleep-for` (see Section 22.10 [Waiting], page 471), or `accept-process-output` (see Section 40.9.4 [Accepting Output], page 1107). Emacs also allows sentinels to run when the command loop is reading input. `delete-process` calls the sentinel when it terminates a running process.

Emacs does not keep a queue of multiple reasons to call the sentinel of one process; it records just the current status and the fact that there has been a change. Therefore two changes in status, coming in quick succession, can call the sentinel just once. However, process termination will always run the sentinel exactly once. This is because the process status can't change again after termination.

Emacs explicitly checks for output from the process before running the process sentinel. Once the sentinel runs due to process termination, no further output can arrive from the process.

A sentinel that writes the output into the buffer of the process should check whether the buffer is still alive. If it tries to insert into a dead buffer, it will get an error. If the buffer is dead, `(buffer-name (process-buffer` *`process`*`))` returns `nil`.

Quitting is normally inhibited within a sentinel—otherwise, the effect of typing *`C-g`* at command level or to quit a user command would be unpredictable. If you want to permit

quitting inside a sentinel, bind `inhibit-quit` to `nil`. In most cases, the right way to do this is with the macro `with-local-quit`. See Section 22.11 [Quitting], page 472.

If an error happens during execution of a sentinel, it is caught automatically, so that it doesn't stop the execution of whatever programs was running when the sentinel was started. However, if `debug-on-error` is non-`nil`, errors are not caught. This makes it possible to use the Lisp debugger to debug the sentinel. See Section 19.1 [Debugger], page 334. If an error is caught, Emacs pauses for `process-error-pause-time` seconds so that the user sees the error. See Section 40.4 [Asynchronous Processes], page 1089.

While a sentinel is running, the process sentinel is temporarily set to `nil` so that the sentinel won't run recursively. For this reason it is not possible for a sentinel to specify a new sentinel.

Note that Emacs automatically saves and restores the match data while executing sentinels. See Section 35.6 [Match Data], page 1014.

`set-process-sentinel` *process sentinel* [Function]

This function associates *sentinel* with *process*. If *sentinel* is `nil`, then the process will have the default sentinel, which inserts a message in the process's buffer when the process status changes.

Changes in process sentinels take effect immediately—if the sentinel is slated to be run but has not been called yet, and you specify a new sentinel, the eventual call to the sentinel will use the new one.

```
(defun msg-me (process event)
   (princ
     (format "Process: %s had the event '%s'" process event)))
(set-process-sentinel (get-process "shell") 'msg-me)
     ⇒ msg-me
(kill-process (get-process "shell"))
     ⊣ Process: #<process shell> had the event 'killed'
     ⇒ #<process shell>
```

`process-sentinel` *process* [Function]

This function returns the sentinel of *process*.

In case a process status changes need to be passed to several sentinels, you can use `add-function` to combine an existing sentinel with a new one. See Section 13.12 [Advising Functions], page 252.

`waiting-for-user-input-p` [Function]

While a sentinel or filter function is running, this function returns non-`nil` if Emacs was waiting for keyboard input from the user at the time the sentinel or filter function was called, or `nil` if it was not.

40.11 Querying Before Exit

When Emacs exits, it terminates all its subprocesses. For subprocesses that run a program, it sends them the `SIGHUP` signal; connections are simply closed. Because subprocesses may be doing valuable work, Emacs normally asks the user to confirm that it is ok to terminate them. Each process has a query flag, which, if non-`nil`, says that Emacs should ask for confirmation before exiting and thus killing that process. The default for the query flag is `t`, meaning *do* query.

`process-query-on-exit-flag` *process* [Function]
: This returns the query flag of *process*.

`set-process-query-on-exit-flag` *process flag* [Function]
: This function sets the query flag of *process* to *flag*. It returns *flag*.

  Here is an example of using `set-process-query-on-exit-flag` on a shell process to avoid querying:

```
(set-process-query-on-exit-flag (get-process "shell") nil)
     ⇒ nil
```

`confirm-kill-processes` [User Option]
: If this user option is set to `t` (the default), then Emacs asks for confirmation before killing processes on exit. If it is `nil`, Emacs kills processes without confirmation, i.e., the query flag of all processes is ignored.

40.12 Accessing Other Processes

In addition to accessing and manipulating processes that are subprocesses of the current Emacs session, Emacs Lisp programs can also access other processes. We call these *system processes*, to distinguish them from Emacs subprocesses.

Emacs provides several primitives for accessing system processes. Not all platforms support these primitives; on those which don't, these primitives return `nil`.

`list-system-processes` [Function]
: This function returns a list of all the processes running on the system. Each process is identified by its PID, a numerical process ID that is assigned by the OS and distinguishes the process from all the other processes running on the same machine at the same time.

  If `default-directory` points to a remote host, processes of that host are returned.

`process-attributes` *pid* [Function]
: This function returns an alist of attributes for the process specified by its process ID *pid*. Each association in the alist is of the form `(`*`key`*` . `*`value`*`)`, where *key* designates the attribute and *value* is the value of that attribute. The various attribute *key*s that this function can return are listed below. Not all platforms support all of these attributes; if an attribute is not supported, its association will not appear in the returned alist.

  If `default-directory` points to a remote host, *pid* is regarded as process of that host.

  `euid` The effective user ID of the user who invoked the process. The corresponding *value* is a number. If the process was invoked by the same user who runs the current Emacs session, the value is identical to what `user-uid` returns (see Section 42.4 [User Identification], page 1274).

  `user` User name corresponding to the process's effective user ID, a string.

  `egid` The group ID of the effective user ID, a number.

  `group` Group name corresponding to the effective user's group ID, a string.

`comm` The name of the command that runs in the process. This is a string that usually specifies the name of the executable file of the process, without the leading directories. However, some special system processes can report strings that do not correspond to an executable file of a program.

`state` The state code of the process. This is a short string that encodes the scheduling state of the process. Here's a list of the most frequently seen codes:

- `"D"` uninterruptible sleep (usually I/O)
- `"R"` running
- `"S"` interruptible sleep (waiting for some event)
- `"T"` stopped, e.g., by a job control signal
- `"Z"` zombie: a process that terminated, but was not reaped by its parent

For the full list of the possible states, see the manual page of the `ps` command.

`ppid` The process ID of the parent process, a number.

`pgrp` The process group ID of the process, a number.

`sess` The session ID of the process. This is a number that is the process ID of the process's *session leader*.

`ttname` A string that is the name of the process's controlling terminal. On Unix and GNU systems, this is normally the file name of the corresponding terminal device, such as `/dev/pts65`.

`tpgid` The numerical process group ID of the foreground process group that uses the process's terminal.

`minflt` The number of minor page faults caused by the process since its beginning. (Minor page faults are those that don't involve reading from disk.)

`majflt` The number of major page faults caused by the process since its beginning. (Major page faults require a disk to be read, and are thus more expensive than minor page faults.)

`cminflt`
`cmajflt` Like `minflt` and `majflt`, but include the number of page faults for all the child processes of the given process.

`utime` Time spent by the process in the user context, for running the application's code. The corresponding *value* is a Lisp timestamp (see Section 42.5 [Time of Day], page 1275).

`stime` Time spent by the process in the system (kernel) context, for processing system calls. The corresponding *value* is a Lisp timestamp.

`time` The sum of `utime` and `stime`. The corresponding *value* is a Lisp timestamp.

`cutime`
`cstime`
`ctime` Like `utime`, `stime`, and `time`, but include the times of all the child processes of the given process.

`pri` The numerical priority of the process.

`nice` The *nice value* of the process, a number. (Processes with smaller nice values get scheduled more favorably.)

`thcount` The number of threads in the process.

`start` The time when the process was started, as a Lisp timestamp.

`etime` The time elapsed since the process started, as a Lisp timestamp.

`vsize` The virtual memory size of the process, measured in kilobytes.

`rss` The size of the process's *resident set*, the number of kilobytes occupied by the process in the machine's physical memory.

`pcpu` The percentage of the CPU time used by the process since it started. The corresponding *value* is a nonnegative floating-point number. Although in theory the number can exceed 100 on a multicore platform, it is usually less than 100 because Emacs is typically single-threaded.

`pmem` The percentage of the total physical memory installed on the machine used by the process's resident set. The value is a floating-point number between 0 and 100.

`args` The command-line with which the process was invoked. This is a string in which individual command-line arguments are separated by blanks; whitespace characters that are embedded in the arguments are quoted as appropriate for the system's shell: escaped by backslash characters on GNU and Unix, and enclosed in double quote characters on Windows. Thus, this command-line string can be directly used in primitives such as `shell-command`.

40.13 Transaction Queues

You can use a *transaction queue* to communicate with a subprocess using transactions. First use `tq-create` to create a transaction queue communicating with a specified process. Then you can call `tq-enqueue` to send a transaction.

`tq-create` *process* [Function]
This function creates and returns a transaction queue communicating with *process*. The argument *process* should be a subprocess capable of sending and receiving streams of bytes. It may be a child process, or it may be a TCP connection to a server, possibly on another machine.

`tq-enqueue` *queue question regexp closure fn* **&optional** *delay-question* [Function]
This function sends a transaction to queue *queue*. Specifying the queue has the effect of specifying the subprocess to talk to.

The argument *question* is the outgoing message that starts the transaction. The argument *fn* is the function to call when the corresponding answer comes back; it is called with two arguments: *closure*, and the answer received.

The argument *regexp* is a regular expression that should match text at the end of the entire answer, but nothing before; that's how `tq-enqueue` determines where the answer ends.

If the argument *delay-question* is non-`nil`, delay sending this question until the process has finished replying to any previous questions. This produces more reliable results with some processes.

`tq-close` *queue* [Function]

Shut down transaction queue *queue*, waiting for all pending transactions to complete, and then terminate the connection or child process.

Transaction queues are implemented by means of a filter function. See Section 40.9.2 [Filter Functions], page 1104.

40.14 Network Connections

Emacs Lisp programs can open stream (TCP) and datagram (UDP) network connections (see Section 40.16 [Datagrams], page 1118) to other processes on the same machine or other machines. A network connection is handled by Lisp much like a subprocess, and is represented by a process object. However, the process you are communicating with is not a child of the Emacs process, has no process ID, and you can't kill it or send it signals. All you can do is send and receive data. `delete-process` closes the connection, but does not kill the program at the other end; that program must decide what to do about closure of the connection.

Lisp programs can listen for connections by creating network servers. A network server is also represented by a kind of process object, but unlike a network connection, the network server never transfers data itself. When it receives a connection request, it creates a new network connection to represent the connection just made. (The network connection inherits certain information, including the process plist, from the server.) The network server then goes back to listening for more connection requests.

Network connections and servers are created by calling `make-network-process` with an argument list consisting of keyword/argument pairs, for example `:server t` to create a server process, or `:type 'datagram` to create a datagram connection. See Section 40.17 [Low-Level Network], page 1118, for details. You can also use the `open-network-stream` function described below.

To distinguish the different types of processes, the `process-type` function returns the symbol `network` for a network connection or server, `serial` for a serial port connection, `pipe` for a pipe connection, or `real` for a real subprocess.

The `process-status` function returns `open`, `closed`, `connect`, `stop`, or `failed` for network connections. For a network server, the status is always `listen`. Except for `stop`, none of those values is possible for a real subprocess. See Section 40.6 [Process Information], page 1095.

You can stop and resume operation of a network process by calling `stop-process` and `continue-process`. For a server process, being stopped means not accepting new connections. (Up to 5 connection requests will be queued for when you resume the server; you can increase this limit, unless it is imposed by the operating system—see the `:server` keyword of `make-network-process`, Section 40.17.1 [Network Processes], page 1118.) For a network stream connection, being stopped means not processing input (any arriving input waits until you resume the connection). For a datagram connection, some number of packets may be queued but input may be lost. You can use the function `process-command` to determine whether a network connection or server is stopped; a non-`nil` value means yes.

Emacs can create encrypted network connections, using the built-in support for the GnuTLS Transport Layer Security Library; see the GnuTLS project page (`https://www.gnu.org/software/gnutls/`). If your Emacs was compiled with GnuTLS support, the function `gnutls-available-p` is defined and returns non-`nil`. For more details, see Section "Overview" in *The Emacs-GnuTLS manual*. The `open-network-stream` function can transparently handle the details of creating encrypted connections for you, using whatever support is available.

`open-network-stream` *name buffer host service* **&rest** *parameters* [Function]

This function opens a TCP connection, with optional encryption, and returns a process object that represents the connection.

The *name* argument specifies the name for the process object. It is modified as necessary to make it unique.

The *buffer* argument is the buffer to associate with the connection. Output from the connection is inserted in the buffer, unless you specify your own filter function to handle the output. If *buffer* is `nil`, it means that the connection is not associated with any buffer.

The arguments *host* and *service* specify where to connect to; *host* is the host name (a string), and *service* is the name of a defined network service (a string) or a port number (an integer like `80` or an integer string like `"80"`).

The remaining arguments *parameters* are keyword/argument pairs that are mainly relevant to encrypted connections:

`:nowait` *boolean*
: If non-`nil`, try to make an asynchronous connection.

`:noquery` *query-flag*
: Initialize the process query flag to *query-flag*. See Section 40.11 [Query Before Exit], page 1110.

`:coding` *coding*
: Use this to set the coding systems used by the network process, in preference to binding `coding-system-for-read` or `coding-system-for-write`. See Section 40.17.1 [Network Processes], page 1118, for details.

`:type` *type*
: The type of connection. Options are:

  `plain` An ordinary, unencrypted connection.

`tls`
`ssl` A TLS (Transport Layer Security) connection.

`nil`
`network` Start with a plain connection, and if parameters '`:success`' and '`:capability-command`' are supplied, try to upgrade to an encrypted connection via STARTTLS. If that fails, retain the unencrypted connection.

`starttls` As for `nil`, but if STARTTLS fails, drop the connection.

`shell` A shell connection.

`:always-query-capabilities` *boolean*
If non-`nil`, always ask for the server's capabilities, even when doing a '`plain`' connection.

`:capability-command` *capability-command*
Command to query the host capabilities. This can either be a string (which will then be sent verbatim to the server) or a function (called with a single parameter: the "greeting" from the server when connecting) that should return a string.

`:end-of-command` *regexp*
`:end-of-capability` *regexp*
Regular expression matching the end of a command, or the end of the command *capability-command*. The latter defaults to the former.

`:starttls-function` *function*
Function of one argument (the response to *capability-command*), which returns either `nil` or the command to activate STARTTLS, if supported.

`:success` *regexp*
Regular expression matching a successful STARTTLS negotiation.

`:use-starttls-if-possible` *boolean*
If non-`nil`, do opportunistic STARTTLS upgrades even if Emacs doesn't have built-in TLS support.

`:warn-unless-encrypted` *boolean*
If non-`nil`, and `:return-value` is also non-`nil`, Emacs will warn if the connection isn't encrypted. This is useful for protocols like IMAP and the like, where most users would expect the network traffic to be encrypted.

`:client-certificate` *list-or-t*
Either a list of the form (*key-file* *cert-file*), naming the certificate key file and certificate file itself, or `t`, meaning to query `auth-source` for this information (see Section "auth-source" in *Emacs auth-source Library*). Only used for TLS or STARTTLS. To enable automatic queries of `auth-source` when `:client-certificate` is not specified customize `network-stream-use-client-certificates` to `t`.

`:return-list` *cons-or-nil*
: The return value of this function. If omitted or `nil`, return a process object. Otherwise, a cons of the form (*process-object* . *plist*), where *plist* can include the following keywords:

  `:greeting` *string-or-nil*
  : If non-`nil`, the greeting string returned by the host.

  `:capabilities` *string-or-nil*
  : If non-`nil`, the host's capability string.

  `:type` *symbol*
  : The connection type: '`plain`' or '`tls`'.

`:shell-command` *string-or-nil*
: If the connection `type` is `shell`, this parameter will be interpreted as a format-spec string (see Section 4.8 [Custom Format Strings], page 70) that will be executed to make the connection. The specs available are '`%s`' for the host name and '`%p`' for the port number. For instance, if you want to first ssh to '`gateway`' before making a plain connection, then this parameter's value could be something like '`ssh gateway nc %s %p`'.

40.15 Network Servers

You create a server by calling `make-network-process` (see Section 40.17.1 [Network Processes], page 1118) with `:server t`. The server will listen for connection requests from clients. When it accepts a client connection request, that creates a new network connection, itself a process object, with the following parameters:

- The connection's process name is constructed by concatenating the server process's *name* with a client identification string. The client identification string for an IPv4 connection looks like '`<a.b.c.d:p>`', which represents an address and port number. Otherwise, it is a unique number in brackets, as in '`<nnn>`'. The number is unique for each connection in the Emacs session.
- If the server has a non-default filter, the connection process does not get a separate process buffer; otherwise, Emacs creates a new buffer for the purpose. The buffer name is the server's buffer name or process name, concatenated with the client identification string.

  The server's process buffer value is never used directly, but the log function can retrieve it and use it to log connections by inserting text there.
- The communication type and the process filter and sentinel are inherited from those of the server. The server never directly uses its filter and sentinel; their sole purpose is to initialize connections made to the server.
- The connection's process contact information is set according to the client's addressing information (typically an IP address and a port number). This information is associated with the `process-contact` keywords `:host`, `:service`, `:remote`.
- The connection's local address is set up according to the port number used for the connection.
- The client process's plist is initialized from the server's plist.

40.16 Datagrams

A *datagram* connection communicates with individual packets rather than streams of data. Each call to `process-send` sends one datagram packet (see Section 40.7 [Input to Processes], page 1099), and each datagram received results in one call to the filter function.

The datagram connection doesn't have to talk with the same remote peer all the time. It has a *remote peer address* which specifies where to send datagrams to. Each time an incoming datagram is passed to the filter function, the peer address is set to the address that datagram came from; that way, if the filter function sends a datagram, it will go back to that place. You can specify the remote peer address when you create the datagram connection using the `:remote` keyword. You can change it later on by calling `set-process-datagram-address`.

`process-datagram-address` *process* [Function]
: If *process* is a datagram connection or server, this function returns its remote peer address.

`set-process-datagram-address` *process address* [Function]
: If *process* is a datagram connection or server, this function sets its remote peer address to *address*.

40.17 Low-Level Network Access

You can also create network connections by operating at a lower level than that of `open-network-stream`, using `make-network-process`.

40.17.1 `make-network-process`

The basic function for creating network connections and network servers is `make-network-process`. It can do either of those jobs, depending on the arguments you give it.

`make-network-process` **&rest** *args* [Function]
: This function creates a network connection or server and returns the process object that represents it. The arguments *args* are a list of keyword/argument pairs. Omitting a keyword is always equivalent to specifying it with value `nil`, except for `:coding`, `:filter-multibyte`, and `:reuseaddr`. Here are the meaningful keywords (those corresponding to network options are listed in the following section):

  :name *name*
  : Use the string *name* as the process name. It is modified if necessary to make it unique.

  :type *type* Specify the communication type. A value of `nil` specifies a stream connection (the default); `datagram` specifies a datagram connection; `seqpacket` specifies a sequenced packet stream connection. Both connections and servers can be of these types.

  :server *server-flag*
  : If *server-flag* is non-`nil`, create a server. Otherwise, create a connection. For a stream type server, *server-flag* may be an integer, which then specifies the length of the queue of pending connections to the server. The default queue length is 5.

:host *host* Specify the host to connect to. *host* should be a host name or Internet address, as a string, or the symbol `local` to specify the local host. If you specify *host* for a server, it must specify a valid address for the local host, and only clients connecting to that address will be accepted. When using `local`, by default IPv4 will be used, specify a *family* of `ipv6` to override this. To listen on all interfaces, specify an address of '`"0.0.0.0"`' for IPv4 or '`"::"`' for IPv6. Note that on some operating systems, listening on '`"::"`' will also listen on IPv4, so attempting to then listen separately on IPv4 will result in `EADDRINUSE` errors ('`"Address already in use"`').

:service *service*
service specifies a port number to connect to; or, for a server, the port number to listen on. It should be a service name like '`"https"`' that translates to a port number, or an integer like '`443`' or an integer string like '`"443"`' that specifies the port number directly. For a server, it can also be `t`, which means to let the system select an unused port number.

:family *family*
family specifies the address (and protocol) family for communication. `nil` means determine the proper address family automatically for the given *host* and *service*. `local` specifies a Unix socket, in which case *host* is ignored. `ipv4` and `ipv6` specify to use IPv4 and IPv6, respectively.

:use-external-socket *use-external-socket*
If *use-external-socket* is non-`nil` use any sockets passed to Emacs on invocation instead of allocating one. This is used by the Emacs server code to allow on-demand socket activation. If Emacs wasn't passed a socket, this option is silently ignored.

:local *local-address*
For a server process, *local-address* is the address to listen on. It overrides *family*, *host* and *service*, so you might as well not specify them.

:remote *remote-address*
For a connection, *remote-address* is the address to connect to. It overrides *family*, *host* and *service*, so you might as well not specify them.

For a datagram server, *remote-address* specifies the initial setting of the remote datagram address.

The format of *local-address* or *remote-address* depends on the address family:

- An IPv4 address is represented as a five-element vector of four 8-bit integers and one 16-bit integer `[a b c d p]` corresponding to numeric IPv4 address *a.b.c.d* and port number *p*.
- An IPv6 address is represented as a nine-element vector of 16-bit integers `[a b c d e f g h p]` corresponding to numeric IPv6 address *a*:*b*:*c*:*d*:*e*:*f*:*g*:*h* and port number *p*.
- A local address is represented as a string, which specifies the address in the local address space.

- An unsupported-family address is represented by a cons `(f . av)`, where *f* is the family number and *av* is a vector specifying the socket address using one element per address data byte. Do not rely on this format in portable code, as it may depend on implementation defined constants, data sizes, and data structure alignment.

:nowait *bool*
: If *bool* is non-`nil` for a stream connection, return without waiting for the connection to complete. When the connection succeeds or fails, Emacs will call the sentinel function, with a second argument matching `"open"` (if successful) or `"failed"`. The default is to block, so that `make-network-process` does not return until the connection has succeeded or failed.

  If you're setting up an asynchronous TLS connection, you have to also provide the `:tls-parameters` parameter (see below).

  Depending on the capabilities of Emacs, how asynchronous `:nowait` is may vary. The three elements that may (or may not) be done asynchronously are domain name resolution, socket setup, and (for TLS connections) TLS negotiation.

  Many functions that interact with process objects, (for instance, `process-datagram-address`) rely on them at least having a socket before they can return a useful value. These functions will block until the socket has achieved the desired status. The recommended way of interacting with asynchronous sockets is to place a sentinel on the process, and not try to interact with it before it has changed status to '`"run"`'. That way, none of these functions will block.

:tls-parameters
: When opening a TLS connection, this should be where the first element is the TLS type (which should either be `gnutls-x509pki` or `gnutls-anon`, and the remaining elements should form a keyword list acceptable for `gnutls-boot`. (This keyword list can be obtained from the `gnutls-boot-parameters` function.) The TLS connection will then be negotiated after completing the connection to the host.

:stop *stopped*
: If *stopped* is non-`nil`, start the network connection or server in the stopped state.

:buffer *buffer*
: Use *buffer* as the process buffer.

:coding *coding*
: Use *coding* as the coding system for this process. To specify different coding systems for decoding data from the connection and for encoding data sent to it, specify `(decoding . encoding)` for *coding*.

  If you don't specify this keyword at all, the default is to determine the coding systems from the data.

:noquery *query-flag*
: Initialize the process query flag to *query-flag*. See Section 40.11 [Query Before Exit], page 1110.

:filter *filter*
: Initialize the process filter to *filter*.

:filter-multibyte *multibyte*
: If *multibyte* is non-`nil`, strings given to the process filter are multibyte, otherwise they are unibyte. The default is `t`.

:sentinel *sentinel*
: Initialize the process sentinel to *sentinel*.

:log *log*
: Initialize the log function of a server process to *log*. The log function is called each time the server accepts a network connection from a client. The arguments passed to the log function are *server*, *connection*, and *message*; where *server* is the server process, *connection* is the new process for the connection, and *message* is a string describing what has happened.

:plist *plist*
: Initialize the process plist to *plist*.

The original argument list, modified with the actual connection information, is available via the `process-contact` function.

40.17.2 Network Options

The following network options can be specified when you create a network process. Except for `:reuseaddr`, you can also set or modify these options later, using `set-network-process-option`.

For a server process, the options specified with `make-network-process` are not inherited by the client connections, so you will need to set the necessary options for each child connection as it is created.

:bindtodevice *device-name*
: If *device-name* is a non-empty string identifying a network interface name (see `network-interface-list`), only handle packets received on that interface. If *device-name* is `nil` (the default), handle packets received on any interface.

  Using this option may require special privileges on some systems.

:broadcast *broadcast-flag*
: If *broadcast-flag* is non-`nil` for a datagram process, the process will receive datagram packet sent to a broadcast address, and be able to send packets to a broadcast address. This is ignored for a stream connection.

:dontroute *dontroute-flag*
: If *dontroute-flag* is non-`nil`, the process can only send to hosts on the same network as the local host.

:keepalive *keepalive-flag*
: If *keepalive-flag* is non-`nil` for a stream connection, enable exchange of low-level keep-alive messages.

:linger *linger-arg*
: If *linger-arg* is non-`nil`, wait for successful transmission of all queued packets on the connection before it is deleted (see `delete-process`). If *linger-arg* is an integer, it specifies the maximum time in seconds to wait for queued packets to be sent before closing the connection. The default is `nil`, which means to discard unsent queued packets when the process is deleted.

:oobinline *oobinline-flag*
: If *oobinline-flag* is non-`nil` for a stream connection, receive out-of-band data in the normal data stream. Otherwise, ignore out-of-band data.

:priority *priority*
: Set the priority for packets sent on this connection to the integer *priority*. The interpretation of this number is protocol specific; such as setting the TOS (type of service) field on IP packets sent on this connection. It may also have system dependent effects, such as selecting a specific output queue on the network interface.

:reuseaddr *reuseaddr-flag*
: If *reuseaddr-flag* is non-`nil` (the default) for a stream server process, allow this server to reuse a specific port number (see `:service`), unless another process on this host is already listening on that port. If *reuseaddr-flag* is `nil`, there may be a period of time after the last use of that port (by any process on the host) where it is not possible to make a new server on that port.

`set-network-process-option` *process option value* **&optional** *no-error* [Function]
: This function sets or modifies a network option for network process *process*. The accepted options and values are as for `make-network-process`. If *no-error* is non-`nil`, this function returns `nil` instead of signaling an error if *option* is not a supported option. If the function successfully completes, it returns `t`.

  The current setting of an option is available via the `process-contact` function.

40.17.3 Testing Availability of Network Features

To test for the availability of a given network feature, use `featurep` like this:

```
(featurep 'make-network-process '(keyword value))
```

The result of this form is `t` if it works to specify *keyword* with value *value* in `make-network-process`. Here are some of the *keyword—value* pairs you can test in this way.

`(:nowait t)`
: Non-`nil` if non-blocking connect is supported.

`(:type datagram)`
: Non-`nil` if datagrams are supported.

`(:family local)`
: Non-`nil` if local (a.k.a. “UNIX domain”) sockets are supported.

`(:family ipv6)`
: Non-`nil` if IPv6 is supported.

`(:service t)`
: Non-`nil` if the system can select the port for a server.

To test for the availability of a given network option, use `featurep` like this:

```
(featurep 'make-network-process 'keyword)
```

The accepted *keyword* values are `:bindtodevice`, etc. For the complete list, see Section 40.17.2 [Network Options], page 1121. This form returns non-`nil` if that particular network option is supported by `make-network-process` (or `set-network-process-option`).

40.18 Misc Network Facilities

These additional functions are useful for creating and operating on network connections. Note that they are supported only on some systems.

`network-interface-list` **&optional** *full family* [Function]

This function returns a list describing the network interfaces of the machine you are using. The value is an alist whose elements have the form `(ifname . address)`. *ifname* is a string naming the interface, *address* has the same form as the *local-address* and *remote-address* arguments to `make-network-process`, i.e. a vector of integers. By default both IPv4 and IPv6 addresses are returned if possible.

Optional argument *full* non-`nil` means to instead return a list of one or more elements of the form `(ifname addr bcast netmask)`. *ifname* is a non-unique string naming the interface. *addr*, *bcast*, and *netmask* are vectors of integers detailing the IP address, broadcast address, and network mask.

Optional argument *family* specified as symbol `ipv4` or `ipv6` restricts the returned information to IPv4 and IPv6 addresses respectively, independently of the value of *full*. Specifying `ipv6` when IPv6 support is not available will result in an error being signaled.

Some examples:

```
(network-interface-list) ⇒
(("vmnet8" .
  [172 16 76 1 0])
 ("vmnet1" .
  [172 16 206 1 0])
 ("lo0" .
  [65152 0 0 0 0 0 0 1 0])
 ("lo0" .
  [0 0 0 0 0 0 0 1 0])
 ("lo0" .
  [127 0 0 1 0]))
(network-interface-list t) ⇒
(("vmnet8"
  [172 16 76 1 0]
  [172 16 76 255 0]
  [255 255 255 0 0])
 ("vmnet1"
```

```
 [172 16 206 1 0]
 [172 16 206 255 0]
 [255 255 255 0 0])
("lo0"
 [65152 0 0 0 0 0 0 1 0]
 [65152 0 0 0 65535 65535 65535 65535 0]
 [65535 65535 65535 65535 0 0 0 0 0])
("lo0"
 [0 0 0 0 0 0 0 1 0]
 [0 0 0 0 0 0 0 1 0]
 [65535 65535 65535 65535 65535 65535 65535 65535 0])
("lo0"
 [127 0 0 1 0]
 [127 255 255 255 0]
 [255 0 0 0 0]))
```

`network-interface-info` *ifname* [Function]

This function returns information about the network interface named *ifname*. The value is a list of the form `(addr bcast netmask hwaddr flags)`.

addr The Internet protocol address.

bcast The broadcast address.

netmask The network mask.

hwaddr The layer 2 address (Ethernet MAC address, for instance).

flags The current flags of the interface.

Note that this function returns only IPv4 information.

`format-network-address` *address* **&optional** *omit-port* [Function]

This function converts the Lisp representation of a network address to a string.

A five-element vector `[a b c d p]` represents an IPv4 address *a.b.c.d* and port number *p*. `format-network-address` converts that to the string `"a.b.c.d:p"`.

A nine-element vector `[a b c d e f g h p]` represents an IPv6 address along with a port number. `format-network-address` converts that to the string `"[a:b:c:d:e:f:g:h]:p"`.

If the vector does not include the port number, *p*, or if *omit-port* is non-`nil`, the result does not include the `:p` suffix.

`network-lookup-address-info` *name* **&optional** *family hints* [Function]

This function perform hostname lookups on *name*, which is expected to be an ASCII-only string, otherwise it signals an error. Call `puny-encode-domain` on *name* first if you wish to lookup internationalized hostnames.

If successful, this function returns a list of Lisp representations of network addresses (see Section 40.17.1 [Network Processes], page 1118, for a description of the format), otherwise return `nil`. In the latter case, it also logs an error message hopefully explaining what went wrong.

By default, this function attempts both IPv4 and IPv6 lookups. The optional argument *family* controls this behavior, specifying the symbol `ipv4` or `ipv6` restricts lookups to IPv4 and IPv6 respectively.

If optional argument *hints* is `numeric`, the function treats the *name* as a numerical IP address (and does not perform DNS lookups). This can be used to check whether a string is a valid numerical representation of an IP address, or to convert a numerical string to its canonical representation. e.g.

```
(network-lookup-address-info "127.1" 'ipv4 'numeric)
    ⇒ ([127 0 0 1 0])

(network-lookup-address-info "::1" nil 'numeric)
    ⇒ ([0 0 0 0 0 0 0 1 0])
```

Be warned that there are some surprising valid forms, especially for IPv4, e.g '`0xe3010203`' and '`0343.1.2.3`' are both valid, as are '`0`' and '`1`' (but they are invalid for IPv6).

40.19 Communicating with Serial Ports

Emacs can communicate with serial ports. For interactive use, *`M-x serial-term`* opens a terminal window. In a Lisp program, `make-serial-process` creates a process object.

The serial port can be configured at run-time, without having to close and re-open it. The function `serial-process-configure` lets you change the speed, bytesize, and other parameters. In a terminal window created by `serial-term`, you can click on the mode line for configuration.

A serial connection is represented by a process object, which can be used in a similar way to a subprocess or network process. You can send and receive data, and configure the serial port. A serial process object has no process ID, however, and you can't send signals to it, and the status codes are different from other types of processes. `delete-process` on the process object or `kill-buffer` on the process buffer close the connection, but this does not affect the device connected to the serial port.

The function `process-type` returns the symbol `serial` for a process object representing a serial port connection.

Serial ports are available on GNU/Linux, Unix, and MS Windows systems.

`serial-term` *port speed* **&optional** *line-mode* [Command]

Start a terminal-emulator for a serial port in a new buffer. *port* is the name of the serial port to connect to. For example, this could be `/dev/ttyS0` on Unix. On MS Windows, this could be `COM1`, or `\\.\COM10` (double the backslashes in Lisp strings).

speed is the speed of the serial port in bits per second. 9600 is a common value. The buffer is in Term mode; see Section "Term Mode" in *The GNU Emacs Manual*, for the commands to use in that buffer. You can change the speed and the configuration in the mode line menu. If *line-mode* is non-`nil`, `term-line-mode` is used; otherwise `term-raw-mode` is used.

`make-serial-process` **&rest** *args* [Function]

This function creates a process and a buffer. Arguments are specified as keyword/argument pairs. Here's the list of the meaningful keywords, with the first two (*port* and *speed*) being mandatory:

`:port` *port*
: This is the name of the serial port. On Unix and GNU systems, this is a file name such as `/dev/ttyS0`. On Windows, this could be `COM1`, or `\\.\COM10` for ports higher than `COM9` (double the backslashes in Lisp strings).

`:speed` *speed*
: The speed of the serial port in bits per second. This function calls `serial-process-configure` to handle the speed; see the following documentation of that function for more details.

`:name` *name*
: The name of the process. If *name* is not given, *port* will serve as the process name as well.

`:buffer` *buffer*
: The buffer to associate with the process. The value can be either a buffer or a string that names a buffer. Process output goes at the end of that buffer, unless you specify an output stream or filter function to handle the output. If *buffer* is not given, the process buffer's name is taken from the value of the `:name` keyword.

`:coding` *coding*
: If *coding* is a symbol, it specifies the coding system used for both reading and writing for this process. If *coding* is a cons (*decoding* . *encoding*), *decoding* is used for reading, and *encoding* is used for writing. If not specified, the default is to determine the coding systems from the data itself.

`:noquery` *query-flag*
: Initialize the process query flag to *query-flag*. See Section 40.11 [Query Before Exit], page 1110. The flags defaults to `nil` if unspecified.

`:stop` *bool*
: Start process in the stopped state if *bool* is non-`nil`. In the stopped state, a serial process does not accept incoming data, but you can send outgoing data. The stopped state is cleared by `continue-process` and set by `stop-process`.

`:filter` *filter*
: Install *filter* as the process filter.

`:sentinel` *sentinel*
: Install *sentinel* as the process sentinel.

`:plist` *plist*
: Install *plist* as the initial plist of the process.

`:bytesize`
`:parity`
`:stopbits`
`:flowcontrol`
: These are handled by `serial-process-configure`, which is called by `make-serial-process`.

The original argument list, possibly modified by later configuration, is available via the function `process-contact`.

Here is an example:

```
(make-serial-process :port "/dev/ttyS0" :speed 9600)
```

`serial-process-configure` **&rest** *args* [Function]

This function configures a serial port connection. Arguments are specified as keyword/argument pairs. Attributes that are not given are re-initialized from the process's current configuration (available via the function `process-contact`), or set to reasonable default values. The following arguments are defined:

`:process` *process*
`:name` *name*
`:buffer` *buffer*
`:port` *port*
: Any of these arguments can be given to identify the process that is to be configured. If none of these arguments is given, the current buffer's process is used.

`:speed` *speed*
: The speed of the serial port in bits per second, a.k.a. *baud rate*. The value can be any number, but most serial ports work only at a few defined values between 1200 and 115200, with 9600 being the most common value. If *speed* is `nil`, the function ignores all other arguments and does not configure the port. This may be useful for special serial ports such as Bluetooth-to-serial converters, which can only be configured through '`AT`' commands sent through the connection. The value of `nil` for *speed* is valid only for connections that were already opened by a previous call to `make-serial-process` or `serial-term`.

`:bytesize` *bytesize*
: The number of bits per byte, which can be 7 or 8. If *bytesize* is not given or `nil`, it defaults to 8.

`:parity` *parity*
: The value can be `nil` (don't use parity), the symbol `odd` (use odd parity), or the symbol `even` (use even parity). If *parity* is not given, it defaults to no parity.

`:stopbits` *stopbits*
: The number of stopbits used to terminate a transmission of each byte. *stopbits* can be 1 or 2. If *stopbits* is not given or `nil`, it defaults to 1.

`:flowcontrol` *flowcontrol*
: The type of flow control to use for this connection, which is either `nil` (don't use flow control), the symbol `hw` (use RTS/CTS hardware flow control), or the symbol `sw` (use XON/XOFF software flow control). If *flowcontrol* is not given, it defaults to no flow control.

Internally, `make-serial-process` calls `serial-process-configure` for the initial configuration of the serial port.

40.20 Packing and Unpacking Byte Arrays

This section describes how to pack and unpack arrays of bytes, usually for binary network protocols. These functions convert byte arrays to alists, and vice versa. The byte array can be represented as a unibyte string or as a vector of integers, while the alist associates symbols either with fixed-size objects or with recursive sub-alists. To use the functions referred to in this section, load the `bindat` library.

Conversion from byte arrays to nested alists is also known as *deserializing* or *unpacking*, while going in the opposite direction is also known as *serializing* or *packing*.

40.20.1 Describing Data Layout

To control unpacking and packing, you write a *data layout specification*, also called a *Bindat type expression*. This can be a *base type* or a *composite type* made of several fields, where the specification controls the length of each field to be processed, and how to pack or unpack it. We normally keep bindat type values in variables whose names end in `-bindat-spec`; that kind of name is automatically recognized as risky (see Section 12.12 [File Local Variables], page 213).

`bindat-type` **&rest** *type* [Macro]
: Creates a Bindat type *value* object according to the Bindat type *expression type*.

A field's *type* describes the size (in bytes) of the object that the field represents and, in the case of multibyte fields, how the bytes are ordered within the field. The two possible orderings are *big endian* (also known as "network byte ordering") and *little endian*. For instance, the number `#x23cd` (decimal 9165) in big endian would be the two bytes `#x23` `#xcd`; and in little endian, `#xcd` `#x23`. Here are the possible type values:

`u8`
`byte` Unsigned byte, with length 1.

`uint` *bitlen* `&optional` *le*
: Unsigned integer in network byte order (big-endian), with *bitlen* bits. *bitlen* has to be a multiple of 8. If *le* is non-`nil`, then use little-endian byte order.

`sint` *bitlen le*
: Signed integer in network byte order (big-endian), with *bitlen* bits. *bitlen* has to be a multiple of 8. If *le* is non-`nil`, then use little-endian byte order.

`str` *len* Unibyte string (see Section 34.1 [Text Representations], page 966) of length *len* bytes. When packing, the first *len* bytes of the input string are copied to the packed output. If the input string is shorter than *len*, the remaining bytes will be

null (zero) unless a pre-allocated string was provided to `bindat-pack`, in which case the remaining bytes are left unmodified. If the input string is multibyte with only ASCII and `eight-bit` characters, it is converted to unibyte before it is packed; other multibyte strings signal an error. When unpacking, any null bytes in the packed input string will appear in the unpacked output.

`strz &optional` *len*
: If *len* is not provided, this is a variable-length null-terminated unibyte string (see Section 34.1 [Text Representations], page 966). When packing into `strz`, the entire input string is copied to the packed output followed by a null (zero) byte. (If pre-allocated string is provided for packing into `strz`, that pre-allocated string should have enough space for the additional null byte appended to the output string contents, see Section 40.20.2 [Bindat Functions], page 1130). The length of the packed output is the length of the input string plus one (for the null terminator). The input string must not contain any null bytes. If the input string is multibyte with only ASCII and `eight-bit` characters, it is converted to unibyte before it is packed; other multibyte strings signal an error. When unpacking a `strz`, the resulting output string will contain all bytes up to (but excluding) the null byte that terminated the input string.

  If *len* is provided, `strz` behaves the same as `str`, but with a couple of differences:

  - When packing, a null terminator is written after the packed input string if the number of characters in the input string is less than *len*.
  - When unpacking, the first null byte encountered in the packed string is interpreted as the terminating byte, and it and all subsequent bytes are excluded from the result of the unpacking.

    Caution: The packed output will not be null-terminated unless the input string is shorter than *len* bytes or it contains a null byte within the first *len* bytes.

`vec` *len* `[`*type*`]`
: Vector of *len* elements. The type of the elements is given by *type*, defaulting to bytes. The *type* can be any Bindat type expression.

`repeat` *len* `[`*type*`]`
: Like `vec`, but it unpacks to and packs from lists, whereas `vec` unpacks to vectors.

`bits` *len*
: List of bits that are set to 1 in *len* bytes. The bytes are taken in big-endian order, and the bits are numbered starting with `8 * `*len*` − 1` and ending with zero. For example: `bits 2` unpacks `#x28 #x1c` to `(2 3 4 11 13)` and `#x1c #x28` to `(3 5 10 11 12)`.

`fill` *len*
: *len* bytes used as a mere filler. In packing, these bytes are left unchanged, which normally means they remain zero. When unpacking, this just returns `nil`.

`align` *len*
: Same as `fill` except the number of bytes is that needed to skip to the next multiple of *len* bytes.

`type` *exp*
: This lets you refer to a type indirectly: *exp* is a Lisp expression which should return a Bindat type *value*.

`unit` *exp* This is a trivial type which uses up 0 bits of space. *exp* describes the value returned when we try to "unpack" such a field.

`struct` *fields...*
Composite type made of several fields. Every field is of the form `(`*name type*`)` where *type* can be any Bindat type expression. *name* can be `_` when the field's value does not deserve to be named, as is often the case for `align` and `fill` fields. When the context makes it clear that this is a Bindat type expression, the symbol `struct` can be omitted.

In the types above, *len* and *bitlen* are given as an integer specifying the number of bytes (or bits) in the field. When the length of a field is not fixed, it typically depends on the value of preceding fields. For this reason, the length *len* does not have to be a constant but can be any Lisp expression and it can refer to the value of previous fields via their name.

For example, the specification of a data layout where a leading byte gives the size of a subsequent vector of 16 bit integers could be:

```
(bindat-type
  (len      u8)
  (payload  vec (1+ len) uint 16))
```

40.20.2 Functions to Unpack and Pack Bytes

In the following documentation, *type* refers to a Bindat type value as returned from `bindat-type`, *raw* to a byte array, and *struct* to an alist representing unpacked field data.

`bindat-unpack` *type raw* **&optional** *idx* [Function]
This function unpacks data from the unibyte string or byte array *raw* according to *type*. Normally, this starts unpacking at the beginning of the byte array, but if *idx* is non-`nil`, it specifies a zero-based starting position to use instead.

The value is an alist or nested alist in which each element describes one unpacked field.

`bindat-get-field` *struct* **&rest** *name* [Function]
This function selects a field's data from the nested alist *struct*. Usually *struct* was returned by `bindat-unpack`. If *name* corresponds to just one argument, that means to extract a top-level field value. Multiple *name* arguments specify repeated lookup of sub-structures. An integer *name* acts as an array index.

For example, `(bindat-get-field` *struct* `a b 2 c)` means to find field `c` in the third element of subfield `b` of field `a`. (This corresponds to *struct*`.a.b[2].c` in the C programming language syntax.)

Although packing and unpacking operations change the organization of data (in memory), they preserve the data's *total length*, which is the sum of all the fields' lengths, in bytes. This value is not generally inherent in either the specification or alist alone; instead, both pieces of information contribute to its calculation. Likewise, the length of a string or array being unpacked may be longer than the data's total length as described by the specification.

`bindat-length` *type struct* [Function]
This function returns the total length of the data in *struct*, according to *type*.

`bindat-pack` *type struct* **&optional** *raw idx* [Function]

This function returns a byte array packed according to *type* from the data in the alist *struct*. It normally creates and fills a new byte array starting at the beginning. However, if *raw* is non-`nil`, it specifies a pre-allocated unibyte string or vector to pack into. If *idx* is non-`nil`, it specifies the starting offset for packing into *raw*.

When pre-allocating, you should make sure `(length raw)` meets or exceeds the total length to avoid an out-of-range error.

`bindat-ip-to-string` *ip* [Function]

Convert the Internet address vector *ip* to a string in the usual dotted notation.

```
(bindat-ip-to-string [127 0 0 1])
     ⇒ "127.0.0.1"
```

40.20.3 Advanced data layout specifications

Bindat type expressions are not limited to the types described earlier. They can also be arbitrary Lisp forms returning Bindat type expressions. For example, the type below describes data which can either contain a 24-bit error code or a vector of bytes:

```
(bindat-type
  (len      u8)
  (payload  . (if (zerop len) (uint 24) (vec (1- len)))))
```

Furthermore, while composite types are normally unpacked to (and packed from) association lists, this can be changed via the use of the following special keyword arguments:

`:unpack-val` *exp*

When the list of fields ends with this keyword argument, then the value returned when unpacking is the value of *exp* instead of the standard alist. *exp* can refer to all the previous fields by their name.

`:pack-val` *exp*

If a field's type is followed by this keyword argument, then the value packed into this field is returned by *exp* instead of being extracted from the alist.

`:pack-var` *name*

If the list of fields is preceded by this keyword argument, then all the subsequent `:pack-val` arguments can refer to the overall value to pack into this composite type via the variable named *name*.

For example, one could describe a 16-bit signed integer as follows:

```
(defconst sint16-bindat-spec
  (let* ((max (ash 1 15))
         (wrap (+ max max)))
    (bindat-type :pack-var v
                 (n uint 16 :pack-val (if (< v 0) (+ v wrap) v))
                 :unpack-val (if (>= n max) (- n wrap) n))))
```

Which would then behave as follows:

```
(bindat-pack sint16-bindat-spec -8)
     ⇒ "\377\370"
```

```
(bindat-unpack sint16-bindat-spec "\300\100")
     ⇒ -16320
```

Finally, you can define new Bindat type forms to use in Bindat type expressions with `bindat-defmacro`:

`bindat-defmacro` *name args* **&rest** *body* [Macro]

Define a new Bindat type expression named *name* and taking arguments *args*. Its behavior follows that of `defmacro`, which the important difference that the new forms can only be used within Bindat type expressions.

41 Emacs Display

This chapter describes a number of features related to the display that Emacs presents to the user.

41.1 Refreshing the Screen

The function `redraw-frame` clears and redisplays the entire contents of a given frame (see Chapter 30 [Frames], page 787). This is useful if the screen is corrupted.

`redraw-frame` **&optional** *frame* [Function]
: This function clears and redisplays frame *frame*. If *frame* is omitted or `nil`, it redraws the selected frame.

Even more powerful is `redraw-display`:

`redraw-display` [Command]
: This function clears and redisplays all visible frames.

In Emacs, processing user input takes priority over redisplay. If you call these functions when input is available, they don't redisplay immediately, but the requested redisplay does happen eventually—after all the input has been processed.

On text terminals, suspending and resuming Emacs normally also refreshes the screen. Some terminal emulators record separate contents for display-oriented programs such as Emacs and for ordinary sequential display. If you are using such a terminal, you might want to inhibit the redisplay on resumption.

`no-redraw-on-reenter` [User Option]
: This variable controls whether Emacs redraws the entire screen after it has been suspended and resumed. Non-`nil` means there is no need to redraw, `nil` means redrawing is needed. The default is `nil`.

41.2 Forcing Redisplay

Emacs normally tries to redisplay the screen whenever it waits for input. With the following function, you can request an immediate attempt to redisplay, in the middle of Lisp code, without actually waiting for input.

`redisplay` **&optional** *force* [Function]
: This function tries immediately to redisplay. The optional argument *force*, if non-`nil`, forces the redisplay to be performed, instead of being preempted if input is pending.

  The function returns `t` if it actually tried to redisplay, and `nil` otherwise. A value of `t` does not mean that redisplay proceeded to completion; it could have been preempted by newly arriving input.

Although `redisplay` tries immediately to redisplay, it does not change how Emacs decides which parts of its frame(s) to redisplay. By contrast, the following function adds certain windows to the pending redisplay work (as if their contents had completely changed), but does not immediately try to perform redisplay.

`force-window-update` **&optional** *object* [Function]
This function forces some or all windows to be updated the next time Emacs does a redisplay. If *object* is a window, that window is to be updated. If *object* is a buffer or buffer name, all windows displaying that buffer are to be updated. If *object* is `nil` (or omitted), all windows are to be updated.

This function does not do a redisplay immediately; Emacs does that as it waits for input, or when the function `redisplay` is called.

`pre-redisplay-function` [Variable]
A function run just before redisplay. It is called with one argument, the set of windows to be redisplayed. The set can be `nil`, meaning only the selected window, or `t`, meaning all the windows.

`pre-redisplay-functions` [Variable]
This hook is run just before redisplay. It is called once in each window that is about to be redisplayed, with `current-buffer` set to the buffer displayed in that window.

41.3 Truncation

When a line of text extends beyond the right edge of a window, Emacs can *continue* the line (make it wrap to the next screen line), or *truncate* the line (limit it to one screen line). The additional screen lines used to display a long text line are called *continuation* lines. Continuation is not the same as filling; continuation happens on the screen only, not in the buffer contents, and it breaks a line precisely at the right margin, not at a word boundary. See Section 33.11 [Filling], page 901.

On a graphical display, tiny arrow images in the window fringes indicate truncated and continued lines (see Section 41.13 [Fringes], page 1193). On a text terminal, and on a graphical display when `fringe-mode` was turned off, a '`$`' in the rightmost column of the window indicates truncation; a '`\`' on the rightmost column indicates a line that wraps. (The display table can specify alternate characters to use for this; see Section 41.23.2 [Display Tables], page 1248).

Since wrapping and truncation of text contradict each other, Emacs turns off line truncation when wrapping is requested, and vice versa.

`truncate-lines` [User Option]
If this buffer-local variable is non-`nil`, lines that extend beyond the right edge of the window are truncated; otherwise, they are continued. As a special exception, the variable `truncate-partial-width-windows` takes precedence in *partial-width* windows (i.e., windows that do not occupy the entire frame width).

`truncate-partial-width-windows` [User Option]
This variable controls line truncation in *partial-width* windows. A partial-width window is one that does not occupy the entire frame width (see Section 29.7 [Splitting Windows], page 710). If the value is `nil`, line truncation is determined by the variable `truncate-lines` (see above). If the value is an integer *n*, lines are truncated if the partial-width window has fewer than *n* columns, regardless of the value of `truncate-lines`; if the partial-width window has *n* or more columns, line truncation is determined by `truncate-lines`. For any other non-`nil` value, lines are truncated in every partial-width window, regardless of the value of `truncate-lines`.

When horizontal scrolling (see Section 29.23 [Horizontal Scrolling], page 770) is in use in a window, that forces truncation.

`wrap-prefix` [Variable]
: If this buffer-local variable is non-`nil`, it defines a *wrap prefix* which Emacs displays at the start of every continuation line. (If lines are truncated, `wrap-prefix` is never used.) Its value may be a string or an image (see Section 41.16.4 [Other Display Specs], page 1207), or a stretch of whitespace such as specified by the `:width` or `:align-to` display properties (see Section 41.16.2 [Specified Space], page 1205). The value is interpreted in the same way as a `display` text property. See Section 41.16 [Display Property], page 1203.

  A wrap prefix may also be specified for regions of text, using the `wrap-prefix` text or overlay property. This takes precedence over the `wrap-prefix` variable. See Section 33.19.4 [Special Properties], page 927.

`line-prefix` [Variable]
: If this buffer-local variable is non-`nil`, it defines a *line prefix* which Emacs displays at the start of every non-continuation line. Its value may be a string or an image (see Section 41.16.4 [Other Display Specs], page 1207), or a stretch of whitespace such as specified by the `:width` or `:align-to` display properties (see Section 41.16.2 [Specified Space], page 1205). The value is interpreted in the same way as a `display` text property. See Section 41.16 [Display Property], page 1203.

  A line prefix may also be specified for regions of text using the `line-prefix` text or overlay property. This takes precedence over the `line-prefix` variable. See Section 33.19.4 [Special Properties], page 927.

41.4 The Echo Area

The *echo area* is used for displaying error messages (see Section 11.7.3 [Errors], page 179), for messages made with the `message` primitive, and for echoing keystrokes. It is not the same as the minibuffer, despite the fact that the minibuffer appears (when active) in the same place on the screen as the echo area. See Section "The Minibuffer" in *The GNU Emacs Manual*.

Apart from the functions documented in this section, you can print Lisp objects to the echo area by specifying `t` as the output stream. See Section 20.4 [Output Streams], page 376.

41.4.1 Displaying Messages in the Echo Area

This section describes the standard functions for displaying messages in the echo area.

`message` *format-string* **&rest** *arguments* [Function]
: This function displays a message in the echo area. *format-string* is a format string, and *arguments* are the objects for its format specifications, like in the `format-message` function (see Section 4.7 [Formatting Strings], page 66). The resulting formatted string is displayed in the echo area; if it contains `face` text properties, it is displayed with the specified faces (see Section 41.12 [Faces], page 1168). The string is also added to the `*Messages*` buffer, but without text properties (see Section 41.4.3 [Logging Messages], page 1141).

Typically grave accent and apostrophe in the format translate to matching curved quotes, e.g., `"Missing ‘%s’"` might result in `"Missing ‘foo’"`. See Section 25.4 [Text Quoting Style], page 602, for how to influence or inhibit this translation.

In batch mode, the message is printed to the standard error stream, followed by a newline.

When `inhibit-message` is non-`nil`, no message will be displayed in the echo area, it will only be logged to ‘`*Messages*`’.

If *format-string* is `nil` or the empty string, `message` clears the echo area; if the echo area has been expanded automatically, this brings it back to its normal size. If the minibuffer is active, this brings the minibuffer contents back onto the screen immediately.

```
(message "Reverting `%s'..." (buffer-name))
 ⊣ Reverting ‘subr.el’...
⇒ "Reverting ‘subr.el’..."

---------- Echo Area ----------
Reverting ‘subr.el’...
---------- Echo Area ----------
```

To automatically display a message in the echo area or in a pop-buffer, depending on its size, use `display-message-or-buffer` (see below).

Warning: If you want to use your own string as a message verbatim, don't just write `(message` *`string`*`)`. If *string* contains ‘`%`’, ‘`` ` ``’, or ‘`'`’ it may be reformatted, with undesirable results. Instead, use `(message "%s"` *`string`*`)`.

The following facilities allow users and Lisp programs to control how echo-area messages are displayed.

`set-message-function` [Variable]

If this variable is non-`nil`, it should be a function of one argument, the text of a message to display in the echo area. That function will be called by `message` and related functions. If the function returns `nil`, the message is displayed in the echo area as usual. If the function returns a string, that string is displayed in the echo area *instead* of the original message. If the function returns any other non-`nil` value, that means the message was already handled, so `message` will not display anything in the echo area.

The default value calls `set-minibuffer-message`, described below.

`clear-message-function` [Variable]

If this variable is non-`nil`, it should be a function of no arguments; `message` and related functions call it when their argument message is `nil` or the empty string, to clear the echo area.

Usually this function is called when the next input event arrives after displaying an echo-area message. The function is expected to clear the message displayed by its counterpart function specified by `set-message-function`, but doesn't have to. If the function wants the echo area to remain uncleared, it should return the symbol `dont-clear-message`; any other value will result in the echo area being cleared.

The default value is the function that clears the message displayed in an active minibuffer.

`set-message-functions` [User Option]
The value of this user option is a list of functions to be called for handling display of echo-area messages. Each function is called with one argument, the text of the message to display. If the function returns a string, that string replaces the original message, and the next function in the list is called with the new message text. If the function returns `nil`, the next function in the list is called with the same text; if the last function in the list returns `nil`, the message text is displayed in the echo area. If the function returns a non-`nil` value that is not a string, the message is considered to be handled, and no further functions in the list are called.

The three useful functions to be put in the list that is the value of this option are described below.

`set-minibuffer-message` *message* [Function]
This function displays *message* in the echo-area when the minibuffer is not active, and at the end of the minibuffer when the minibuffer is active. However, if the text shown in the active minibuffer has the `minibuffer-message` text property (see Section 33.19.4 [Special Properties], page 927) on some character, the message will be displayed before the first character having that property.

This function is by default the only member of the list in `set-message-functions`.

`inhibit-message` *message* [Function]
If an echo-area *message* matches any regexp in the list that is the value of the user option `inhibit-message-regexps`, this function suppresses the display of that message and returns a non-`nil` value that is not a string. Thus, if this function is in the list `set-message-functions`, the rest of the functions in the list will not be called when *message* matches the regexps in `inhibit-message-regexps`. To ensure a matching *message* will never be displayed, make this function be the first element of the list in `set-message-functions`.

`set-multi-message` *message* [Function]
This function accumulates several echo-area messages emitted one after another, and returns them as a single string in which individual messages are separated by newlines. Up to `multi-message-max` recent messages can be accumulated. The accumulated messages are discarded when more than `multi-message-timeout` seconds have elapsed since the time the first message was emitted.

`inhibit-message` [Variable]
When this variable is non-`nil`, `message` and related functions will not display any messages in the Echo Area. Echo-area messages are still logged in the `*Messages*` buffer, though.

`with-temp-message` *message* **&rest** *body* [Macro]
This construct displays a message in the echo area temporarily, during the execution of *body*. It displays *message*, executes *body*, then returns the value of the last body form while restoring the previous echo area contents.

`message-or-box` *format-string* **&rest** *arguments* [Function]
: This function displays a message like `message`, but may display it in a dialog box instead of the echo area. If this function is called in a command that was invoked using the mouse—more precisely, if `last-nonmenu-event` (see Section 22.5 [Command Loop Info], page 437) is either `nil` or a list—then it uses a dialog box or pop-up menu to display the message. Otherwise, it uses the echo area. (This is the same criterion that `y-or-n-p` uses to make a similar decision; see Section 21.7 [Yes-or-No Queries], page 414.)

  You can force use of the mouse or of the echo area by binding `last-nonmenu-event` to a suitable value around the call.

`message-box` *format-string* **&rest** *arguments* [Function]
: This function displays a message like `message`, but uses a dialog box (or a pop-up menu) whenever that is possible. If it is impossible to use a dialog box or pop-up menu, because the terminal does not support them, then `message-box` uses the echo area, like `message`.

`display-message-or-buffer` *message* **&optional** *buffer-name action frame* [Function]
: This function displays the message *message*, which may be either a string or a buffer. If it is shorter than the maximum height of the echo area, as defined by `max-mini-window-height`, it is displayed in the echo area, using `message`. Otherwise, `display-buffer` is used to show it in a pop-up buffer.

  Returns either the string shown in the echo area, or when a pop-up buffer is used, the window used to display it.

  If *message* is a string, then the optional argument *buffer-name* is the name of the buffer used to display it when a pop-up buffer is used, defaulting to `*Message*`. In the case where *message* is a string and displayed in the echo area, it is not specified whether the contents are inserted into the buffer anyway.

  The optional arguments *action* and *frame* are as for `display-buffer`, and only used if a buffer is displayed.

`current-message` [Function]
: This function returns the message currently being displayed in the echo area, or `nil` if there is none.

41.4.2 Reporting Operation Progress

When an operation can take a while to finish, you should inform the user about the progress it makes. This way the user can estimate remaining time and clearly see that Emacs is busy working, not hung. A convenient way to do this is to use a *progress reporter*.

Here is a working example that does nothing useful:

```
(let ((progress-reporter
       (make-progress-reporter "Collecting mana for Emacs..."
                               0  500)))
  (dotimes (k 500)
    (sit-for 0.01)
    (progress-reporter-update progress-reporter k))
  (progress-reporter-done progress-reporter))
```

`make-progress-reporter` *message* **&optional** *min-value max-value current-value min-change min-time* [Function]

This function creates and returns a progress reporter object, which you will use as an argument for the other functions listed below. The idea is to precompute as much data as possible to make progress reporting very fast.

When this progress reporter is subsequently used, it will display *message* in the echo area, followed by progress percentage. *message* is treated as a simple string. If you need it to depend on a filename, for instance, use `format-message` before calling this function.

The arguments *min-value* and *max-value* should be numbers standing for the starting and final states of the operation. For instance, an operation that scans a buffer should set these to the results of `point-min` and `point-max` correspondingly. *max-value* should be greater than *min-value*.

Alternatively, you can set *min-value* and *max-value* to `nil`. In that case, the progress reporter does not report process percentages; it instead displays a "spinner" that rotates a notch each time you update the progress reporter.

If *min-value* and *max-value* are numbers, you can give the argument *current-value* a numerical value specifying the initial progress; if omitted, this defaults to *min-value*.

The remaining arguments control the rate of echo area updates. The progress reporter will wait for at least *min-change* more percents of the operation to be completed before printing next message; the default is one percent. *min-time* specifies the minimum time in seconds to pass between successive prints; the default is 0.2 seconds. (On some operating systems, the progress reporter may handle fractions of seconds with varying precision).

This function calls `progress-reporter-update`, so the first message is printed immediately.

`progress-reporter-update` *reporter* **&optional** *value suffix* [Function]

This function does the main work of reporting progress of your operation. It displays the message of *reporter*, followed by progress percentage determined by *value*. If percentage is zero, or close enough according to the *min-change* and *min-time* arguments, then it is omitted from the output.

reporter must be the result of a call to `make-progress-reporter`. *value* specifies the current state of your operation and must be between *min-value* and *max-value* (inclusive) as passed to `make-progress-reporter`. For instance, if you scan a buffer, then *value* should be the result of a call to `point`.

Optional argument *suffix* is a string to be displayed after *reporter*'s main message and progress text. If *reporter* is a non-numerical reporter, then *value* should be `nil`, or a string to use instead of *suffix*.

This function respects *min-change* and *min-time* as passed to `make-progress-reporter` and so does not output new messages on every invocation. It is thus very fast and normally you should not try to reduce the number of calls to it: resulting overhead will most likely negate your effort.

`progress-reporter-force-update` *reporter* **&optional** *value new-message suffix* [Function]

This function is similar to `progress-reporter-update` except that it prints a message in the echo area unconditionally.

reporter, *value*, and *suffix* have the same meaning as for `progress-reporter-update`. Optional *new-message* allows you to change the message of the *reporter*. Since this function always updates the echo area, such a change will be immediately presented to the user.

`progress-reporter-done` *reporter* [Function]

This function should be called when the operation is finished. It prints the message of *reporter* followed by word '`done`' in the echo area.

You should always call this function and not hope for `progress-reporter-update` to print '`100%`'. Firstly, it may never print it, there are many good reasons for this not to happen. Secondly, '`done`' is more explicit.

`dotimes-with-progress-reporter` (*var count* [*result*]) *reporter-or-message body...* [Macro]

This is a convenience macro that works the same way as `dotimes` does, but also reports loop progress using the functions described above. It allows you to save some typing. The argument *reporter-or-message* can be either a string or a progress reporter object.

You can rewrite the example in the beginning of this subsection using this macro as follows:

```
(dotimes-with-progress-reporter
    (k 500)
    "Collecting some mana for Emacs..."
  (sit-for 0.01))
```

Using a reporter object as the *reporter-or-message* argument is useful if you want to specify the optional arguments in *make-progress-reporter*. For instance, you can write the previous example as follows:

```
(dotimes-with-progress-reporter
    (k 500)
    (make-progress-reporter "Collecting some mana for Emacs..." 0 500 0 1 1.5)
  (sit-for 0.01))
```

`dolist-with-progress-reporter` (*var list* [*result*]) *reporter-or-message body...* [Macro]

This is another convenience macro that works the same way as `dolist` does, but also reports loop progress using the functions described above. As in `dotimes-with-progress-reporter`, `reporter-or-message` can be a progress reporter or a string. You can rewrite the previous example with this macro as follows:

```
(dolist-with-progress-reporter
    (k (number-sequence 0 500))
    "Collecting some mana for Emacs..."
  (sit-for 0.01))
```

`with-delayed-message` (*timeout message*) *body*... [Macro]

Sometimes it's unclear whether an operation will take a long time to execute or not, or it can be inconvenient to implement a progress reporter. This macro can be used in those situations.

```
(with-delayed-message (2 (format "Gathering data for %s" entry))
  (setq data (gather-data entry)))
```

In this example, if the body takes more than two seconds to execute, the message will be displayed. If it takes a shorter time than that, the message won't be displayed. In either case, the body is evaluated as normally, and the return value of the final element in the body is the return value of the macro.

The *message* element is evaluated before *body*, and is always evaluated, whether the message is displayed or not.

41.4.3 Logging Messages in `*Messages*`

Almost all the messages displayed in the echo area are also recorded in the `*Messages*` buffer so that the user can refer back to them. This includes all the messages that are output with `message`. By default, this buffer is read-only and uses the major mode `messages-buffer-mode`. Nothing prevents the user from killing the `*Messages*` buffer, but the next display of a message recreates it. Any Lisp code that needs to access the `*Messages*` buffer directly and wants to ensure that it exists should use the function `messages-buffer`.

`messages-buffer` [Function]

This function returns the `*Messages*` buffer. If it does not exist, it creates it, and switches it to `messages-buffer-mode`.

`message-log-max` [User Option]

This variable specifies how many lines to keep in the `*Messages*` buffer. The value `t` means there is no limit on how many lines to keep. The value `nil` disables message logging entirely. Here's how to display a message and prevent it from being logged:

```
(let (message-log-max)
  (message ...))
```

`messages-buffer-name` [Variable]

This variable has the name of the buffer where messages should be logged to, and defaults to `*Messages*`. Some packages may find it useful to temporarily redirect the output to a different buffer (perhaps to write the buffer out to a log file later), and they can bind this variable to a different buffer name. (Note that this buffer (if it doesn't exist already), will be created and put into `messages-buffer-mode`.)

To make `*Messages*` more convenient for the user, the logging facility combines successive identical messages. It also combines successive related messages for the sake of two cases: question followed by answer, and a series of progress messages.

A question followed by an answer has two messages like the ones produced by `y-or-n-p`: the first is '*`question`*', and the second is '*`question...answer`*'. The first message conveys no additional information beyond what's in the second, so logging the second message discards the first from the log.

A series of progress messages has successive messages like those produced by `make-progress-reporter`. They have the form '*`base...how-far`*', where *base* is the same each time, while *how-far* varies. Logging each message in the series discards the previous one, provided they are consecutive.

The functions `make-progress-reporter` and `y-or-n-p` don't have to do anything special to activate the message log combination feature. It operates whenever two consecutive messages are logged that share a common prefix ending in '`...`'.

41.4.4 Echo Area Customization

These variables control details of how the echo area works.

`cursor-in-echo-area` [Variable]
: This variable controls where the cursor appears when a message is displayed in the echo area. If it is non-`nil`, then the cursor appears at the end of the message. Otherwise, the cursor appears at point—not in the echo area at all.

  The value is normally `nil`; Lisp programs bind it to `t` for brief periods of time.

`echo-area-clear-hook` [Variable]
: This normal hook is run whenever the echo area is cleared—either by `(message nil)` or for any other reason.

`echo-keystrokes` [User Option]
: This variable determines how much time should elapse before command characters echo. Its value must be a number, and specifies the number of seconds to wait before echoing. If the user types a prefix key (such as `C-x`) and then delays this many seconds before continuing, the prefix key is echoed in the echo area. (Once echoing begins in a key sequence, all subsequent characters in the same key sequence are echoed immediately.)

  If the value is zero, then command input is not echoed.

`message-truncate-lines` [Variable]
: Normally, displaying a long message resizes the echo area to display the entire message, wrapping long line as needed. But if the variable `message-truncate-lines` is non-`nil`, long lines of echo-area message are instead truncated to fit the mini-window width.

The variable `max-mini-window-height`, which specifies the maximum height for resizing minibuffer windows, also applies to the echo area (which is really a special use of the minibuffer window; see Section 21.11 [Minibuffer Windows], page 419).

41.5 Reporting Warnings

Warnings are a facility for a program to inform the user of a possible problem, but continue running (as opposed to signaling an error, see Section 11.7.3 [Errors], page 179).

41.5.1 Warning Basics

Every warning is a textual message, which explains the problem for the user, with the associated *severity level* which is a symbol. Here are the supported severity levels, in order of decreasing severity, and their meanings:

`:emergency`
: A problem that will seriously impair Emacs operation soon if the user does not attend to it promptly.

`:error`
: A report about data or circumstances that are inherently wrong.

`:warning`
: A report about data or circumstances that are not inherently wrong, but raise suspicion of a possible problem.

`:debug`
: A report of information that may be useful if the user is currently debugging the Lisp program which issues the warning.

When your program encounters invalid input data, it can either signal a Lisp error by calling `error` or `signal` (see Section 11.7.3.1 [Signaling Errors], page 180) or report a warning with severity `:error`. Signaling a Lisp error is the easiest thing to do, but it means the signaling program cannot continue execution. If you want to take the trouble of implementing a way to continue processing despite the invalid data, then reporting a warning of severity `:error` is the right way of informing the user of the problem. For instance, the Emacs Lisp byte compiler can report an error that way and continue compiling other functions. (If the program signals a Lisp error and then handles it with `condition-case`, the user won't see the error message; reporting that as a warning instead avoids that problem.)

In addition to severity level, each warning has a *warning type* to classify it. The warning type is either a symbol or a list of symbols. If it is a symbol, it should be the custom group that you use for the program's user options; if it is a list, the first element of the list should be that custom group. For example, byte compiler warnings use the warning type `(bytecomp)`. If the warning type is a list, the elements of the list after the first one, which should be arbitrary symbols, represent subcategories of the warning: they will be displayed to the user to better explain the nature of the warning.

`display-warning` *type message* **&optional** *level buffer-name* [Function]
: This function reports a warning, using the string *message* as the warning text and *type* as the warning type. *level* should be the severity level, and defaults to `:warning` if omitted or `nil`.

  buffer-name, if non-`nil`, specifies the name of the buffer for logging the warning message. By default, it is `*Warnings*`.

`lwarn` *type level message* **&rest** *args* [Function]
: This function reports a warning using the value returned by `(format-message` *message args...*`)` as the message text in the `*Warnings*` buffer. In other respects it is equivalent to `display-warning`.

`warn` *message* **&rest** *args* [Function]
: This function reports a warning using the value returned by `(format-message` *message args...*`)` as the message text, `emacs` as the warning type, and `:warning` as the severity level. It exists for compatibility only; we recommend not using it, because you should specify a specific warning type.

41.5.2 Warning Variables

Programs can customize how their warnings appear by binding the variables described in this section.

`warning-levels` [Variable]

This list defines the meaning and severity order of the warning severity levels. Each element defines one severity level, and they are arranged in order of decreasing severity.

Each element has the form (*level string* [*function*]), where *level* is the severity level it defines. *string* specifies the textual description of this level. *string* should use '`%s`' to specify where to put the warning type information, or it can omit the '`%s`' so as not to include that information.

The optional *function*, if non-`nil`, is a function to call with no arguments, to get the user's attention. A notable example is `ding` (see Section 41.24 [Beeping], page 1252).

Normally you should not change the value of this variable.

`warning-prefix-function` [Variable]

If non-`nil`, the value is a function to generate prefix text for warnings. Programs can bind the variable to a suitable function. `display-warning` calls this function with the warnings buffer the current buffer, and the function can insert text into it. That text becomes the beginning of the warning message.

The function is called with two arguments, the severity level and its entry in `warning-levels`. It should return a list to use *instead* of that entry (the value need not be an actual member of `warning-levels`, but it must have the same structure). By constructing this value, the function can change the severity of the warning, or specify different handling for a given severity level.

If the variable's value is `nil`, there's no prefix text, before the warning is displayed, starting with the *string* part of the entry in `warning-levels` corresponding to the warning's level.

`warning-series` [Variable]

Programs can bind this variable to `t` to say that the next warning should begin a series. When several warnings form a series, that means to leave point on the first warning of the series, rather than keep moving it for each warning so that it appears on the last one. The series ends when the local binding of this variable is unbound and `warning-series` becomes `nil` again.

The value can also be a symbol with a function definition. That is equivalent to `t`, except that the next warning will also call the function with no arguments with the warnings buffer the current buffer. The function can, for example, insert text which will serve as a header for the series of warnings.

Once a series has begun, the value of this variable is a marker which points to the buffer position in the warnings buffer of the start of the series.

The variable's normal value is `nil`, which means to handle each warning separately.

`warning-fill-prefix` [Variable]

When this variable is non-`nil`, it specifies a fill prefix to use for filling the text of each warning.

`warning-fill-column` [Variable]
The column at which to fill warnings.

`warning-type-format` [Variable]
This variable specifies the format for displaying the warning type in the warning text. The result of formatting the type this way gets included in the message under the control of the string in the entry in `warning-levels`. The default value is `" (%s)"`. If you bind it to the empty string `""` then the warning type won't appear at all.

41.5.3 Warning Options

These variables are used by users to control what happens when a Lisp program reports a warning.

`warning-minimum-level` [User Option]
This user option specifies the minimum severity level that should be shown immediately to the user, by popping the warnings buffer in some window. The default is `:warning`, which means to show the warning buffer for any warning severity except `:debug`. The warnings of lower severity levels will still be written into the warnings buffer, but the buffer will not be forced onto display.

`warning-minimum-log-level` [User Option]
This user option specifies the minimum severity level that should be logged in the warnings buffer. Warnings of lower severity will be completely ignored: not written to the warnings buffer and not displayed. The default is `:warning`, which means to log warnings of any severity except `:debug`.

`warning-suppress-types` [User Option]
This list specifies which warning types should not be displayed immediately when they occur. Each element of the list should be a list of symbols. If an element of this list has the same elements as the first elements in a warning type, then the warning of that type will not be shown on display by popping the warnings buffer in some window (the warning will still be logged in the warnings buffer).

For example, if the value of this variable is a list like this:

```
((foo) (bar subtype))
```

then warnings whose types are `foo` or `(foo)` or `(foo something)` or `(bar subtype other)` will not be shown to the user.

`warning-suppress-log-types` [User Option]
This list specifies which warning types should be ignored: not logged in the warnings buffer and not shown to the user. The structure and the matching of warning types are the same as for `warning-suppress-types` above.

During startup, Emacs delays showing any warnings until after it loads and processes the site-wide and user's init files (see Section 42.1.1 [Startup Summary], page 1260). Let-binding (see Section 12.3 [Local Variables], page 190) the values of these options around some code in your init files which might emit a warning will therefore not work, because it will not be in effect by the time the warning is actually processed. Thus, if you want to suppress some warnings during startup, change the values of the above options in your init file early

enough, or put those let-binding forms in your `after-init-hook` or `emacs-startup-hook` functions. See Section 42.1.2 [Init File], page 1263.

41.5.4 Delayed Warnings

Sometimes, you may wish to avoid showing a warning while a command is running, and only show it only after the end of the command. You can use the function `delay-warning` for this. Emacs automatically delays any warnings emitted during the early stages of startup, and shows them only after the init files are processed.

`delay-warning` *type message* **&optional** *level buffer-name* [Function]
: This function is the delayed counterpart to `display-warning` (see Section 41.5.1 [Warning Basics], page 1143), and it is called with the same arguments. The warning message is queued into `delayed-warnings-list`.

`delayed-warnings-list` [Variable]
: The value of this variable is a list of warnings to be displayed after the current command has finished. Each element must be a list

```
(type message [level [buffer-name]])
```

with the same form, and the same meanings, as the argument list of `display-warning`. Immediately after running `post-command-hook` (see Section 22.1 [Command Overview], page 424), the Emacs command loop displays all the warnings specified by this variable, then resets the variable to `nil`.

Programs which need to further customize the delayed warnings mechanism can change the variable `delayed-warnings-hook`:

`delayed-warnings-hook` [Variable]
: This is a normal hook which is run by the Emacs command loop, after `post-command-hook`, in order to process and display delayed warnings. Emacs also runs this hook during startup, after loading the site-start and user init files (see Section 42.1.1 [Startup Summary], page 1260), because warnings emitted before that are automatically delayed.

Its default value is a list of two functions:

```
(collapse-delayed-warnings display-delayed-warnings)
```

The function `collapse-delayed-warnings` removes repeated entries from `delayed-warnings-list`. The function `display-delayed-warnings` calls `display-warning` on each of the entries in `delayed-warnings-list`, in turn, and then sets `delayed-warnings-list` to `nil`.

41.6 Invisible Text

You can make characters *invisible*, so that they do not appear on the screen, with the `invisible` property. This can be either a text property (see Section 33.19 [Text Properties], page 919) or an overlay property (see Section 41.9 [Overlays], page 1154). Cursor motion also partly ignores these characters; if the command loop finds that point is inside a range of invisible text after a command, it relocates point to the other side of the text.

In the simplest case, any non-`nil` `invisible` property makes a character invisible. This is the default case—if you don't alter the default value of `buffer-invisibility-spec`,

this is how the `invisible` property works. You should normally use `t` as the value of the `invisible` property if you don't plan to set `buffer-invisibility-spec` yourself.

More generally, you can use the variable `buffer-invisibility-spec` to control which values of the `invisible` property make text invisible. This permits you to classify the text into different subsets in advance, by giving them different `invisible` values, and subsequently make various subsets visible or invisible by changing the value of `buffer-invisibility-spec`.

Controlling visibility with `buffer-invisibility-spec` is especially useful in a program to display the list of entries in a database. It permits the implementation of convenient filtering commands to view just a part of the entries in the database. Setting this variable is very fast, much faster than scanning all the text in the buffer looking for properties to change.

`buffer-invisibility-spec` [Variable]

This variable specifies which kinds of `invisible` properties actually make a character invisible. Setting this variable makes it buffer-local.

`t` A character is invisible if its `invisible` property is non-`nil`. This is the default.

a list Each element of the list specifies a criterion for invisibility; if a character's `invisible` property fits any one of these criteria, the character is invisible. The list can have two kinds of elements:

`atom` A character is invisible if its `invisible` property value is *atom* or if it is a list with *atom* as a member; comparison is done with `eq`.

`(`*`atom`*` . t)`
A character is invisible if its `invisible` property value is *atom* or if it is a list with *atom* as a member; comparison is done with `eq`. Moreover, a sequence of such characters displays as an ellipsis.

Two functions are specifically provided for adding elements to `buffer-invisibility-spec` and removing elements from it.

`add-to-invisibility-spec` *element* [Function]

This function adds the element *element* to `buffer-invisibility-spec`. If `buffer-invisibility-spec` was `t`, it changes to a list, `(t)`, so that text whose `invisible` property is `t` remains invisible.

`remove-from-invisibility-spec` *element* [Function]

This removes the element *element* from `buffer-invisibility-spec`. This does nothing if *element* is not in the list.

A convention for use of `buffer-invisibility-spec` is that a major mode should use the mode's own name as an element of `buffer-invisibility-spec` and as the value of the `invisible` property:

```
;; If you want to display an ellipsis:
```

```
(add-to-invisibility-spec '(my-symbol . t))
;; If you don't want ellipsis:
(add-to-invisibility-spec 'my-symbol)

(overlay-put (make-overlay beginning end)
             'invisible 'my-symbol)

;; When done with the invisibility:
(remove-from-invisibility-spec '(my-symbol . t))
;; Or respectively:
(remove-from-invisibility-spec 'my-symbol)
```

You can check for invisibility using the following function:

`invisible-p` *pos-or-prop* [Function]

If *pos-or-prop* is a marker or number, this function returns a non-`nil` value if the text at that position is currently invisible.

If *pos-or-prop* is any other kind of Lisp object, that is taken to mean a possible value of the `invisible` text or overlay property. In that case, this function returns a non-`nil` value if that value would cause text to become invisible, based on the current value of `buffer-invisibility-spec`.

The return value of this function is `t` if the text would be completely hidden on display, or a non-`nil`, non-`t` value if the text would be replaced by an ellipsis.

Ordinarily, functions that operate on text or move point do not care whether the text is invisible, they process invisible characters and visible characters alike. The user-level line motion commands, such as `next-line`, `previous-line`, ignore invisible newlines if `line-move-ignore-invisible` is non-`nil` (the default), i.e., behave like these invisible newlines didn't exist in the buffer, but only because they are explicitly programmed to do so.

If a command ends with point inside or at the boundary of invisible text, the main editing loop relocates point to one of the two ends of the invisible text. Emacs chooses the direction of relocation so that it is the same as the overall movement direction of the command; if in doubt, it prefers a position where an inserted char would not inherit the `invisible` property. Additionally, if the text is not replaced by an ellipsis and the command only moved within the invisible text, then point is moved one extra character so as to try and reflect the command's movement by a visible movement of the cursor.

Thus, if the command moved point back to an invisible range (with the usual stickiness), Emacs moves point back to the beginning of that range. If the command moved point forward into an invisible range, Emacs moves point forward to the first visible character that follows the invisible text and then forward one more character.

These *adjustments* of point that ended up in the middle of invisible text can be disabled by setting `disable-point-adjustment` to a non-`nil` value. See Section 22.6 [Adjusting Point], page 440.

Incremental search can make invisible overlays visible temporarily and/or permanently when a match includes invisible text. To enable this, the overlay should have a non-`nil` `isearch-open-invisible` property. The property value should be a function to be called

with the overlay as an argument. This function should make the overlay visible permanently; it is used when the match overlaps the overlay on exit from the search.

During the search, such overlays are made temporarily visible by temporarily modifying their invisible and intangible properties. If you want this to be done differently for a certain overlay, give it an `isearch-open-invisible-temporary` property which is a function. The function is called with two arguments: the first is the overlay, and the second is `nil` to make the overlay visible, or `t` to make it invisible again.

41.7 Selective Display

Selective display refers to a pair of related features for hiding certain lines on the screen.

The first variant, explicit selective display, was designed for use in a Lisp program: it controls which lines are hidden by altering the text. This kind of hiding is now obsolete and deprecated; instead you should use the `invisible` property (see Section 41.6 [Invisible Text], page 1146) to get the same effect.

In the second variant, the choice of lines to hide is made automatically based on indentation. This variant is designed to be a user-level feature.

The way you control explicit selective display is by replacing a newline (control-j) with a carriage return (control-m). The text that was formerly a line following that newline is now hidden. Strictly speaking, it is temporarily no longer a line at all, since only newlines can separate lines; it is now part of the previous line.

Selective display does not directly affect editing commands. For example, `C-f` (`forward-char`) moves point unhesitatingly into hidden text. However, the replacement of newline characters with carriage return characters affects some editing commands. For example, `next-line` skips hidden lines, since it searches only for newlines. Modes that use selective display can also define commands that take account of the newlines, or that control which parts of the text are hidden.

When you write a selectively displayed buffer into a file, all the control-m's are output as newlines. This means that when you next read in the file, it looks OK, with nothing hidden. The selective display effect is seen only within Emacs.

`selective-display` [Variable]

This buffer-local variable enables selective display. This means that lines, or portions of lines, may be made hidden.

- If the value of `selective-display` is `t`, then the character control-m marks the start of hidden text; the control-m, and the rest of the line following it, are not displayed. This is explicit selective display.
- If the value of `selective-display` is a positive integer, then lines that start with more than that many columns of indentation are not displayed.

When some portion of a buffer is hidden, the vertical movement commands operate as if that portion did not exist, allowing a single `next-line` command to skip any number of hidden lines. However, character movement commands (such as `forward-char`) do not skip the hidden portion, and it is possible (if tricky) to insert or delete text in a hidden portion.

In the examples below, we show the *display appearance* of the buffer `foo`, which changes with the value of `selective-display`. The *contents* of the buffer do not change.

```
(setq selective-display nil)
     ⇒ nil

---------- Buffer: foo ----------
1 on this column
 2on this column
  3n this column
  3n this column
 2on this column
1 on this column
---------- Buffer: foo ----------

(setq selective-display 2)
     ⇒ 2

---------- Buffer: foo ----------
1 on this column
 2on this column
 2on this column
1 on this column
---------- Buffer: foo ----------
```

`selective-display-ellipses` [User Option]

If this buffer-local variable is non-`nil`, then Emacs displays '`...`' at the end of a line that is followed by hidden text. This example is a continuation of the previous one.

```
(setq selective-display-ellipses t)
     ⇒ t

---------- Buffer: foo ----------
1 on this column
 2on this column ...
 2on this column
1 on this column
---------- Buffer: foo ----------
```

You can use a display table to substitute other text for the ellipsis ('`...`'). See Section 41.23.2 [Display Tables], page 1248.

41.8 Temporary Displays

Temporary displays are used by Lisp programs to put output into a buffer and then present it to the user for perusal rather than for editing. Many help commands use this feature.

`with-output-to-temp-buffer` *buffer-name body...* [Macro]

This function executes the forms in *body* while arranging to insert any output they print into the buffer named *buffer-name*, which is first created if necessary, and put

into Help mode. (See the similar form `with-temp-buffer-window` below.) Finally, the buffer is displayed in some window, but that window is not selected.

If the forms in *body* do not change the major mode in the output buffer, so that it is still Help mode at the end of their execution, then `with-output-to-temp-buffer` makes this buffer read-only at the end, and also scans it for function and variable names to make them into clickable cross-references. See [Tips for Documentation Strings], page 1344, in particular the item on hyperlinks in documentation strings, for more details.

The string *buffer-name* specifies the temporary buffer, which need not already exist. The argument must be a string, not a buffer. The buffer is erased initially (with no questions asked), and it is marked as unmodified after `with-output-to-temp-buffer` exits.

`with-output-to-temp-buffer` binds `standard-output` to the temporary buffer, then it evaluates the forms in *body*. Output using the Lisp output functions within *body* goes by default to that buffer (but screen display and messages in the echo area, although they are "output" in the general sense of the word, are not affected). See Section 20.5 [Output Functions], page 378.

Several hooks are available for customizing the behavior of this construct; they are listed below.

The value of the last form in *body* is returned.

```
---------- Buffer: foo ----------
 This is the contents of foo.
---------- Buffer: foo ----------

(with-output-to-temp-buffer "foo"
    (print 20)
    (print standard-output))
⇒ #<buffer foo>

---------- Buffer: foo ----------

20

#<buffer foo>

---------- Buffer: foo ----------
```

`temp-buffer-show-function` [User Option]

If this variable is non-`nil`, `with-output-to-temp-buffer` calls it as a function to do the job of displaying a help buffer. The function gets one argument, which is the buffer it should display.

It is a good idea for this function to run `temp-buffer-show-hook` just as `with-output-to-temp-buffer` normally would, inside of `save-selected-window` and with the chosen window and buffer selected.

`temp-buffer-setup-hook` [Variable]
: This normal hook is run by `with-output-to-temp-buffer` before evaluating *body*. When the hook runs, the temporary buffer is current. This hook is normally set up with a function to put the buffer in Help mode.

`temp-buffer-show-hook` [Variable]
: This normal hook is run by `with-output-to-temp-buffer` after displaying the temporary buffer. When the hook runs, the temporary buffer is current, and the window it was displayed in is selected.

`with-temp-buffer-window` *buffer-or-name action quit-function body...* [Macro]
: This macro is similar to `with-output-to-temp-buffer`. Like that construct, it executes *body* while arranging to insert any output it prints into the buffer named *buffer-or-name* and displays that buffer in some window. Unlike `with-output-to-temp-buffer`, however, it does not automatically switch that buffer to Help mode.

  The argument *buffer-or-name* specifies the temporary buffer. It can be either a buffer, which must already exist, or a string, in which case a buffer of that name is created, if necessary. The buffer is marked as unmodified and read-only when `with-temp-buffer-window` exits.

  This macro does not call `temp-buffer-show-function`. Rather, it passes the *action* argument to `display-buffer` (see Section 29.13.1 [Choosing Window], page 726) in order to display the buffer.

  The value of the last form in *body* is returned, unless the argument *quit-function* is specified. In that case, it is called with two arguments: the window showing the buffer and the result of *body*. The final return value is then whatever *quit-function* returns.

  This macro uses the normal hooks `temp-buffer-window-setup-hook` and `temp-buffer-window-show-hook` in place of the analogous hooks run by `with-output-to-temp-buffer`.

The two constructs described next are mostly identical to `with-temp-buffer-window` but differ from it as specified:

`with-current-buffer-window` *buffer-or-name action quit-function* **&rest** *body* [Macro]
: This macro is like `with-temp-buffer-window` but unlike that makes the buffer specified by *buffer-or-name* current for running *body*.

A window showing a temporary buffer can be fitted to the size of that buffer using the following mode:

`temp-buffer-resize-mode` [User Option]
: When this minor mode is enabled, windows showing a temporary buffer are automatically resized to fit their buffer's contents.

  A window is resized if and only if it has been specially created for the buffer. In particular, windows that have shown another buffer before are not resized. By default, this mode uses `fit-window-to-buffer` (see Section 29.5 [Resizing Windows],

page 705) for resizing. You can specify a different function by customizing the options `temp-buffer-max-height` and `temp-buffer-max-width` below.

The effect of this option can be overridden by providing a suitable `window-height`, `window-width` or `window-size` action alist entry for `display-buffer` (see Section 29.13.3 [Buffer Display Action Alists], page 733).

`temp-buffer-max-height` [User Option]

This option specifies the maximum height (in lines) of a window displaying a temporary buffer when `temp-buffer-resize-mode` is enabled. It can also be a function to be called to choose the height for such a buffer. It gets one argument, the buffer, and should return a positive integer. At the time the function is called, the window to be resized is selected.

`temp-buffer-max-width` [User Option]

This option specifies the maximum width of a window (in columns) displaying a temporary buffer when `temp-buffer-resize-mode` is enabled. It can also be a function to be called to choose the width for such a buffer. It gets one argument, the buffer, and should return a positive integer. At the time the function is called, the window to be resized is selected.

The following function uses the current buffer for temporary display:

`momentary-string-display` *string position* **&optional** *char message* [Function]

This function momentarily displays *string* in the current buffer at *position*. It has no effect on the undo list or on the buffer's modification status.

The momentary display remains until the next input event. If the next input event is *char*, `momentary-string-display` ignores it and returns. Otherwise, that event remains buffered for subsequent use as input. Thus, typing *char* will simply remove the string from the display, while typing (say) `C-f` will remove the string from the display and later (presumably) move point forward. The argument *char* is a space by default.

The return value of `momentary-string-display` is not meaningful.

If the string *string* does not contain control characters, you can do the same job in a more general way by creating (and then subsequently deleting) an overlay with a `before-string` property. See Section 41.9.2 [Overlay Properties], page 1157.

If *message* is non-`nil`, it is displayed in the echo area while *string* is displayed in the buffer. If it is `nil`, a default message says to type *char* to continue.

In this example, point is initially located at the beginning of the second line:

```
---------- Buffer: foo ----------
This is the contents of foo.
⋆Second line.
---------- Buffer: foo ----------

(momentary-string-display
  "**** Important Message! ****"
  (point) ?\r
  "Type RET when done reading")
⇒ t
```

```
---------- Buffer: foo ----------
This is the contents of foo.
**** Important Message! ****Second line.
---------- Buffer: foo ----------

---------- Echo Area ----------
Type RET when done reading
---------- Echo Area ----------
```

41.9 Overlays

You can use *overlays* to alter the appearance of a buffer's text on the screen, for the sake of presentation features. An overlay is an object that belongs to a particular buffer, and has a specified beginning and end. It also has properties that you can examine and set; these affect the display of the text within the overlaid portion of the buffer.

Editing the text of the buffer adjusts the beginning and end of each overlay so that it stays with the text. When you create the overlay, you can specify whether text inserted at the beginning should be inside the overlay or outside, and likewise for the end of the overlay.

41.9.1 Managing Overlays

This section describes the functions to create, delete and move overlays, and to examine their contents. Overlay changes are not recorded in the buffer's undo list, since the overlays are not considered part of the buffer's contents.

`overlayp` *object* [Function]
: This function returns `t` if *object* is an overlay.

`make-overlay` *start end* **&optional** *buffer front-advance rear-advance* [Function]
: This function creates and returns an overlay that belongs to *buffer* and ranges from *start* to *end*. Both *start* and *end* must specify buffer positions; they may be integers or markers. If *buffer* is omitted, the overlay is created in the current buffer.

  An overlay whose *start* and *end* specify the same buffer position is known as *empty*. A non-empty overlay can become empty if the text between its *start* and *end* is deleted. When that happens, the overlay is by default not deleted, but you can cause it to be deleted by giving it the '`evaporate`' property (see Section 41.9.2 [Overlay Properties], page 1157).

  The arguments *front-advance* and *rear-advance* specify what happens when text is inserted at the beginning (i.e., before *start*) and at the end. If they are both `nil`, the default, then the overlay extends to include any text inserted at the beginning, but not text inserted at the end. If *front-advance* is non-`nil`, text inserted at the beginning of the overlay is excluded from the overlay. If *rear-advance* is non-`nil`, text inserted at the end of the overlay is included in the overlay.

`overlay-start` *overlay* [Function]
: This function returns the position at which *overlay* starts, as an integer.

`overlay-end` *overlay* [Function]
This function returns the position at which *overlay* ends, as an integer.

`overlay-buffer` *overlay* [Function]
This function returns the buffer that *overlay* belongs to. It returns `nil` if *overlay* has been deleted.

`delete-overlay` *overlay* [Function]
This function deletes the specified *overlay*. The overlay continues to exist as a Lisp object, and its property list is unchanged, but it ceases to be attached to the buffer it belonged to, and ceases to have any effect on display.

A deleted overlay is not permanently disconnected. You can give it a position in a buffer again by calling `move-overlay`.

`move-overlay` *overlay start end* **&optional** *buffer* [Function]
This function moves *overlay* to *buffer*, and places its bounds at *start* and *end* in that buffer. Both arguments *start* and *end* must specify buffer positions; they may be integers or markers.

If *buffer* is omitted, *overlay* stays in the same buffer it was already associated with; if *overlay* was previously deleted (and thus isn't associated with any buffer), it goes into the current buffer.

The return value is *overlay*.

This function is the only valid way to change the endpoints of an overlay.

`remove-overlays` **&optional** *start end name value* [Function]
This function removes all the overlays between *start* and *end* whose property *name* has the specified *value*. It can move the endpoints of the overlays in the region, or split them.

If *name* is omitted or `nil`, it means to delete all overlays in the specified region. If *start* and/or *end* are omitted or `nil`, that means the beginning and end of the buffer respectively. Therefore, `(remove-overlays)` removes all the overlays in the current buffer.

`copy-overlay` *overlay* [Function]
This function returns a copy of *overlay*. The copy has the same endpoints and properties as *overlay*. However, the text insertion type for the start of the overlay and for the end of the overlay are set to their default values.

Here are some examples:

```
;; Create an overlay.
(setq foo (make-overlay 1 10))
     ⇒ #<overlay from 1 to 10 in display.texi>
(overlay-start foo)
     ⇒ 1
```

```
(overlay-end foo)
     ⇒ 10
(overlay-buffer foo)
     ⇒ #<buffer display.texi>
;; Give it a property we can check later.
(overlay-put foo 'happy t)
     ⇒ t
;; Verify the property is present.
(overlay-get foo 'happy)
     ⇒ t
;; Move the overlay.
(move-overlay foo 5 20)
     ⇒ #<overlay from 5 to 20 in display.texi>
(overlay-start foo)
     ⇒ 5
(overlay-end foo)
     ⇒ 20
;; Delete the overlay.
(delete-overlay foo)
     ⇒ nil
;; Verify it is deleted.
foo
     ⇒ #<overlay in no buffer>
;; A deleted overlay has no position.
(overlay-start foo)
     ⇒ nil
(overlay-end foo)
     ⇒ nil
(overlay-buffer foo)
     ⇒ nil
;; Undelete the overlay.
(move-overlay foo 1 20)
     ⇒ #<overlay from 1 to 20 in display.texi>
;; Verify the results.
(overlay-start foo)
     ⇒ 1
(overlay-end foo)
     ⇒ 20
(overlay-buffer foo)
     ⇒ #<buffer display.texi>
;; Moving and deleting the overlay does not change its properties.
(overlay-get foo 'happy)
     ⇒ t
```

41.9.2 Overlay Properties

Overlay properties are like text properties in that the properties that alter how a character is displayed can come from either source. But in most respects they are different. See Section 33.19 [Text Properties], page 919, for comparison.

Text properties are considered a part of the text; overlays and their properties are specifically considered not to be part of the text. Thus, copying text between various buffers and strings preserves text properties, but does not try to preserve overlays. Changing a buffer's text properties marks the buffer as modified, while moving an overlay or changing its properties does not. Unlike text property changes, overlay property changes are not recorded in the buffer's undo list.

Since more than one overlay can specify a property value for the same character, Emacs lets you specify a priority value of each overlay. The priority value is used to decide which of the overlapping overlays will "win".

These functions read and set the properties of an overlay:

`overlay-get` *overlay prop* [Function]
: This function returns the value of property *prop* recorded in *overlay*, if any. If *overlay* does not record any value for that property, but it does have a `category` property which is a symbol, that symbol's *prop* property is used. Otherwise, the value is `nil`.

`overlay-put` *overlay prop value* [Function]
: This function sets the value of property *prop* recorded in *overlay* to *value*. It returns *value*.

`overlay-properties` *overlay* [Function]
: This returns a copy of the property list of *overlay*.

See also the function `get-char-property` which checks both overlay properties and text properties for a given character. See Section 33.19.1 [Examining Properties], page 920.

Many overlay properties have special meanings; here is a table of them:

`priority`
: This property's value determines the priority of the overlay. If you want to specify a priority value, use either `nil` (or zero), or a positive integer, or a cons of two values. Any other value triggers undefined behavior.

  The priority matters when two or more overlays cover the same character and both specify the same property with different values; the one whose `priority` value is higher overrides the other. (For the `face` property, the higher priority overlay's value does not completely override the other value; instead, its individual face attributes override the corresponding face attributes of the `face` property whose priority is lower.) If two overlays have the same priority value, and one is "nested" in the other (i.e., covers fewer buffer or string positions), then the inner one will prevail over the outer one. If neither is nested in the other then you should not make assumptions about which overlay will prevail.

  When a Lisp program puts overlays with defined priorities on text that might have overlays without priorities, this could cause undesirable results, because any overlay with a positive priority value will override all the overlays without a priority. Since most Emacs features that use overlays don't specify priorities

for their overlays, integer priorities should be used with care. Instead of using integer priorities and risk overriding other overlays, you can use priority values of the form `(primary . secondary)`, where the *primary* value is used as described above, and *secondary* is the fallback value used when *primary* and the nesting considerations fail to resolve the precedence between overlays. In particular, priority value `(nil . n)`, with *n* a positive integer, allows to have the overlays ordered by priority when necessary without completely overriding other overlays.

Currently, all overlays take priority over text properties.

If you need to put overlays in priority order, use the *sorted* argument of `overlays-at`. See Section 41.9.3 [Finding Overlays], page 1161.

`window` If the `window` property is non-`nil`, then the overlay applies only on that window.

`category` If an overlay has a `category` property, we call it the *category* of the overlay. It should be a symbol. The properties of the symbol serve as defaults for the properties of the overlay.

`face` This property controls the appearance of the text (see Section 41.12 [Faces], page 1168). The value of the property can be the following:

- A face name (a symbol or string).
- An anonymous face: a property list of the form `(keyword value ...)`, where each *keyword* is a face attribute name and *value* is a value for that attribute.
- A list of faces. Each list element should be either a face name or an anonymous face. This specifies a face which is an aggregate of the attributes of each of the listed faces. Faces occurring earlier in the list have higher priority.
- A cons cell of the form `(foreground-color . color-name)` or `(background-color . color-name)`. This specifies the foreground or background color, similar to `(:foreground color-name)` or `(:background color-name)`. This form is supported for backward compatibility only, and should be avoided.

`mouse-face`
This property is used instead of `face` when the mouse is within the range of the overlay. However, Emacs ignores all face attributes from this property that alter the text size (e.g., `:height`, `:weight`, and `:slant`); those attributes are always the same as in the unhighlighted text.

`display` This property activates various features that change the way text is displayed. For example, it can make text appear taller or shorter, higher or lower, wider or narrower, or replaced with an image. See Section 41.16 [Display Property], page 1203.

`help-echo`
If an overlay has a `help-echo` property, then when you move the mouse onto the text in the overlay, Emacs displays a help string in the echo area, or as a tooltip. For details see [Text help-echo], page 928.

`field` Consecutive characters with the same `field` property constitute a *field*. Some motion functions including `forward-word` and `beginning-of-line` stop moving at a field boundary. See Section 33.19.9 [Fields], page 939.

`modification-hooks`
This property's value is a list of functions to be called if any character within the overlay is changed or if text is inserted strictly within the overlay.

The hook functions are called both before and after each change. If the functions save the information they receive, and compare notes between calls, they can determine exactly what change has been made in the buffer text.

When called before a change, each function receives four arguments: the overlay, `nil`, and the beginning and end of the text range to be modified.

When called after a change, each function receives five arguments: the overlay, `t`, the beginning and end of the text range just modified, and the length of the pre-change text replaced by that range. (For an insertion, the pre-change length is zero; for a deletion, that length is the number of characters deleted, and the post-change beginning and end are equal.)

When these functions are called, `inhibit-modification-hooks` is bound to non-`nil`. If the functions modify the buffer, you might want to bind `inhibit-modification-hooks` to `nil`, so as to cause the change hooks to run for these modifications. However, doing this may call your own change hook recursively, so be sure to prepare for that. See Section 33.34 [Change Hooks], page 964.

Text properties also support the `modification-hooks` property, but the details are somewhat different (see Section 33.19.4 [Special Properties], page 927).

`insert-in-front-hooks`
This property's value is a list of functions to be called before and after inserting text right at the beginning of the overlay. The calling conventions are the same as for the `modification-hooks` functions.

`insert-behind-hooks`
This property's value is a list of functions to be called before and after inserting text right at the end of the overlay. The calling conventions are the same as for the `modification-hooks` functions.

`invisible`
The `invisible` property can make the text in the overlay invisible, which means that it does not appear on the screen. See Section 41.6 [Invisible Text], page 1146, for details.

`intangible`
The `intangible` property on an overlay works just like the `intangible` text property. It is obsolete. See Section 33.19.4 [Special Properties], page 927, for details.

`isearch-open-invisible`
This property tells incremental search (see Section "Incremental Search" in *The GNU Emacs Manual*) how to make an invisible overlay visible, permanently, if the final match overlaps it. See Section 41.6 [Invisible Text], page 1146.

`isearch-open-invisible-temporary`
: This property tells incremental search how to make an invisible overlay visible, temporarily, during the search. See Section 41.6 [Invisible Text], page 1146.

`before-string`
: This property's value is a string to add to the display at the beginning of the overlay. The string does not appear in the buffer in any sense—only on the screen. Note that if the text at the beginning of the overlay is made invisible, the string will not be displayed.

`after-string`
: This property's value is a string to add to the display at the end of the overlay. The string does not appear in the buffer in any sense—only on the screen. Note that if the text at the end of the overlay is made invisible, the string will not be displayed.

`line-prefix`
: This property specifies a display spec to prepend to each non-continuation line at display-time. See Section 41.3 [Truncation], page 1134.

`wrap-prefix`
: This property specifies a display spec to prepend to each continuation line at display-time. See Section 41.3 [Truncation], page 1134.

`evaporate`
: If this property is non-`nil`, the overlay is deleted automatically if it becomes empty (i.e., if its length becomes zero). If you give an empty overlay (see Section 41.9.1 [Managing Overlays], page 1154) a non-`nil` `evaporate` property, that deletes it immediately. Note that, unless an overlay has this property, it will not be deleted when the text between its starting and ending positions is deleted from the buffer.

`keymap` If this property is non-`nil`, it specifies a keymap for a portion of the text. This keymap takes precedence over most other keymaps (see Section 23.7 [Active Keymaps], page 490), and it is used when point is within the overlay, where the front- and rear-advance properties define whether the boundaries are considered as being *within* or not.

`local-map`
: The `local-map` property is similar to `keymap` but replaces the buffer's local map rather than augmenting existing keymaps. This also means it has lower precedence than minor mode keymaps.

The `keymap` and `local-map` properties do not affect a string displayed by the `before-string`, `after-string`, or `display` properties. This is only relevant for mouse clicks and other mouse events that fall on the string, since point is never on the string. To bind special mouse events for the string, assign it a `keymap` or `local-map` text property. See Section 33.19.4 [Special Properties], page 927.

41.9.3 Searching for Overlays

`overlays-at` *pos* **&optional** *sorted* [Function]
This function returns a list of all the overlays that cover the character at position *pos* in the current buffer. If *sorted* is non-`nil`, the list is in decreasing order of priority, otherwise it is in no particular order. An overlay contains position *pos* if it begins at or before *pos*, and ends after *pos*.

To illustrate usage, here is a Lisp function that returns a list of the overlays that specify property *prop* for the character at point:

```
(defun find-overlays-specifying (prop)
  (let ((overlays (overlays-at (point)))
        found)
    (while overlays
      (let ((overlay (car overlays)))
        (if (overlay-get overlay prop)
            (setq found (cons overlay found))))
      (setq overlays (cdr overlays)))
    found))
```

`overlays-in` *beg end* [Function]
This function returns a list of the overlays that overlap the region *beg* through *end*. An overlay overlaps with a region if it contains one or more characters in the region; empty overlays (see Section 41.9.1 [Managing Overlays], page 1154) overlap if they are at *beg*, strictly between *beg* and *end*, or at *end* when *end* denotes the position at the end of the accessible part of the buffer.

`next-overlay-change` *pos* [Function]
This function returns the buffer position of the next beginning or end of an overlay, after *pos*. If there is none, it returns `(point-max)`.

`previous-overlay-change` *pos* [Function]
This function returns the buffer position of the previous beginning or end of an overlay, before *pos*. If there is none, it returns `(point-min)`.

As an example, here's a simplified (and inefficient) version of the primitive function `next-single-char-property-change` (see Section 33.19.3 [Property Search], page 924). It searches forward from position *pos* for the next position where the value of a given property `prop`, as obtained from either overlays or text properties, changes.

```
(defun next-single-char-property-change (position prop)
  (save-excursion
    (goto-char position)
    (let ((propval (get-char-property (point) prop)))
      (while (and (not (eobp))
                  (eq (get-char-property (point) prop) propval))
        (goto-char (min (next-overlay-change (point))
                        (next-single-property-change (point) prop)))))
    (point)))
```

41.10 Size of Displayed Text

Since not all characters have the same width, these functions let you check the width of a character. See Section 33.17.1 [Primitive Indent], page 913, and Section 31.2.5 [Screen Lines], page 860, for related functions.

`char-width` *char* [Function]
: This function returns the width in columns of the character *char*, if it were displayed in the current buffer (i.e., taking into account the buffer's display table, if any; see Section 41.23.2 [Display Tables], page 1248). The width of a tab character is usually `tab-width` (see Section 41.23.1 [Usual Display], page 1247).

`char-uppercase-p` *char* [Function]
: Return non-`nil` if *char* is an uppercase character according to Unicode.

`string-width` *string* **&optional** *from to* [Function]
: This function returns the width in columns of the string *string*, if it were displayed in the current buffer and the selected window. Optional arguments *from* and *to* specify the substring of *string* to consider, and are interpreted as in `substring` (see Section 4.3 [Creating Strings], page 55).

  The return value is an approximation: it only considers the values returned by `char-width` for the constituent characters, always takes a tab character as taking `tab-width` columns, ignores display properties and fonts, etc. For these reasons, we recommend using `window-text-pixel-size` or `string-pixel-width`, described below, instead.

`truncate-string-to-width` *string width* **&optional** *start-column padding ellipsis ellipsis-text-property* [Function]
: This function returns a new string that is a truncation of *string* which fits within *width* columns on display.

  If *string* is narrower than *width*, the result is equal to *string*; otherwise excess characters are omitted from the result. If a multi-column character in *string* exceeds the goal *width*, that character is omitted from the result. Thus, the result can sometimes fall short of *width*, but cannot go beyond it.

  The optional argument *start-column* specifies the starting column; it defaults to zero. If this is non-`nil`, then the first *start-column* columns of the string are omitted from the result. If one multi-column character in *string* extends across the column *start-column*, that character is omitted.

  The optional argument *padding*, if non-`nil`, is a padding character added at the beginning and end of the result string, to extend it to exactly *width* columns. The padding character is appended at the end of the result if it falls short of *width*, as many times as needed to reach *width*. It is also prepended at the beginning of the result if a multi-column character in *string* extends across the column *start-column*.

  If *ellipsis* is non-`nil`, it should be a string which will replace the end of *string* when it is truncated. In this case, more characters will be removed from *string* to free enough space for *ellipsis* to fit within *width* columns. However, if the display width of *string* is less than the display width of *ellipsis*, *ellipsis* will not be appended to the result. If *ellipsis* is non-`nil` and not a string, it stands for the value returned by the function `truncate-string-ellipsis`, described below.

  The optional argument *ellipsis-text-property*, if non-`nil`, means hide the excess parts of *string* with a `display` text property (see Section 41.16 [Display Property], page 1203) showing the ellipsis, instead of actually truncating the string.

```
(truncate-string-to-width "\tab\t" 12 4)
     ⇒ "ab"
(truncate-string-to-width "\tab\t" 12 4 ?\s)
     ⇒ "    ab  "
```

This function uses `string-width` and `char-width` to find the suitable truncation point when *string* is too wide, so it suffers from the same basic issues as `string-width` does. In particular, when character composition happens within *string*, the display width of a string could be smaller than the sum of widths of the constituent characters, and this function might return inaccurate results.

`truncate-string-ellipsis` [Function]

This function returns the string to be used as an ellipses in `truncate-string-to-width` and other similar contexts. The value is that of the variable `truncate-string-ellipsis`, if it's non-`nil`, the string with the single character U+2026 HORIZONTAL ELLIPSIS if that character can be displayed on the selected frame, and the string '`...`' otherwise.

The following function returns the size in pixels of text as if it were displayed in a given window. This function is used by `fit-window-to-buffer` and `fit-frame-to-buffer` (see Section 29.5 [Resizing Windows], page 705) to make a window exactly as large as the text it contains.

`window-text-pixel-size` **&optional** *window from to x-limit y-limit mode-lines ignore-line-at-end* [Function]

This function returns the size of the text of *window*'s buffer in pixels. *window* must be a live window and defaults to the selected one. The return value is a cons of the maximum pixel-width of any text line and the maximum pixel-height of all text lines. This function exists to allow Lisp programs to adjust the dimensions of *window* to the buffer text it needs to display, and for other similar situations.

The return value can also optionally (see below) include the buffer position of the first line whose dimensions were measured.

The optional argument *from*, if non-`nil`, specifies the first text position to consider, and defaults to the minimum accessible position of the buffer. If *from* is `t`, it stands for the minimum accessible position that is not a newline character. If *from* is a cons, its `car` specifies a buffer position, and its `cdr` specifies the vertical offset in pixels from that position to the first screen line whose text is to be measured. (The measurement will start from the visual beginning of that screen line.) In that case, the return value will instead be a list of the pixel-width, pixel-height, and the buffer position of the first line that was measured. The optional argument *to*, if non-`nil`, specifies the last text position to consider, and defaults to the maximum accessible position of the buffer. If *to* is `t`, it stands for the maximum accessible position that is not a newline character.

The optional argument *x-limit*, if non-`nil`, specifies the maximum X coordinate beyond which text should be ignored; it is therefore also the largest value of pixel-width that the function can return. If *x-limit* `nil` or omitted, it means to use the pixel-width of *window*'s body (see Section 29.4 [Window Sizes], page 701); this default means that text of truncated lines wider than the window will be ignored. This default is useful

when the caller does not intend to change the width of *window*. Otherwise, the caller should specify here the maximum width *window*'s body may assume; in particular, if truncated lines are expected and their text needs to be accounted for, *x-limit* should be set to a large value. Since calculating the width of long lines can take some time, it's always a good idea to make this argument as small as needed; in particular, if the buffer might contain long lines that will be truncated anyway.

The optional argument *y-limit*, if non-`nil`, specifies the maximum Y coordinate beyond which text is to be ignored; it is therefore also the maximum pixel-height that the function can return. If *y-limit* is `nil` or omitted, it means to consider all the lines of text till the buffer position specified by *to*. Since calculating the pixel-height of a large buffer can take some time, it makes sense to specify this argument; in particular, if the caller does not know the size of the buffer.

The optional argument *mode-lines* `nil` or omitted means to not include the height of the mode-, tab- or header-line of *window* in the return value. If it is either the symbol `mode-line`, `tab-line` or `header-line`, include only the height of that line, if present, in the return value. If it is `t`, include the height of all of these lines, if present, in the return value.

The optional argument *ignore-line-at-end* controls whether or not to count the height of text in *to*'s screen line as part of the returned pixel-height. This is useful if your Lisp program is only interested in the dimensions of text up to and excluding the visual beginning of *to*'s screen line.

`window-text-pixel-size` treats the text displayed in a window as a whole and does not care about the size of individual lines. The following function does.

`window-lines-pixel-dimensions` **&optional** *window first last body inverse left* [Function]

This function calculates the pixel dimensions of each line displayed in the specified *window*. It does so by walking *window*'s current glyph matrix—a matrix storing the glyph (see Section 41.23.4 [Glyphs], page 1250) of each buffer character currently displayed in *window*. If successful, it returns a list of cons pairs representing the x- and y-coordinates of the lower right corner of the last character of each line. Coordinates are measured in pixels from an origin (0, 0) at the top-left corner of *window*. *window* must be a live window and defaults to the selected one.

If the optional argument *first* is an integer, it denotes the index (starting with 0) of the first line of *window*'s glyph matrix to be returned. Note that if *window* has a header line, the line with index 0 is that header line. If *first* is `nil`, the first line to be considered is determined by the value of the optional argument *body*: If *body* is non-`nil`, this means to start with the first line of *window*'s body, skipping any header line, if present. Otherwise, this function will start with the first line of *window*'s glyph matrix, possibly the header line.

If the optional argument *last* is an integer, it denotes the index of the last line of *window*'s glyph matrix that shall be returned. If *last* is `nil`, the last line to be considered is determined by the value of *body*: If *body* is non-`nil`, this means to use the last line of *window*'s body, omitting *window*'s mode line, if present. Otherwise, this means to use the last line of *window* which may be the mode line.

The optional argument *inverse*, if `nil`, means that the y-pixel value returned for any line specifies the distance in pixels from the left edge (body edge if *body* is non-`nil`) of *window* to the right edge of the last glyph of that line. *inverse* non-`nil` means that the y-pixel value returned for any line specifies the distance in pixels from the right edge of the last glyph of that line to the right edge (body edge if *body* is non-`nil`) of *window*. This is useful for determining the amount of slack space at the end of each line.

The optional argument *left*, if non-`nil` means to return the x- and y-coordinates of the lower left corner of the leftmost character on each line. This is the value that should be used for windows that mostly display text from right to left.

If *left* is non-`nil` and *inverse* is `nil`, this means that the y-pixel value returned for any line specifies the distance in pixels from the left edge of the last (leftmost) glyph of that line to the right edge (body edge if *body* is non-`nil`) of *window*. If *left* and *inverse* are both non-`nil`, the y-pixel value returned for any line specifies the distance in pixels from the left edge (body edge if *body* is non-`nil`) of *window* to the left edge of the last (leftmost) glyph of that line.

This function returns `nil` if the current glyph matrix of *window* is not up-to-date which usually happens when Emacs is busy, for example, when processing a command. The value should be retrievable though when this function is run from an idle timer with a delay of zero seconds.

`buffer-text-pixel-size` **&optional** *buffer-or-name window from to x-limit y-limit* [Function]

This is much like `window-text-pixel-size`, but can be used when the buffer isn't shown in a window. (`window-text-pixel-size` is faster when it is, so this function shouldn't be used in that case.)

buffer-or-name must specify a live buffer or the name of a live buffer and defaults to the current buffer. *window* must be a live window and defaults to the selected one; the function will compute the text dimensions as if *buffer* is displayed in *window*. The return value is a cons of the maximum pixel-width of any text line and the pixel-height of all the text lines of the buffer specified by *buffer-or-name*.

The optional arguments *x-limit* and *y-limit* have the same meaning as with `window-text-pixel-size`.

`string-pixel-width` *string* [Function]

This is a convenience function that uses `window-text-pixel-size` to compute the width of *string* (in pixels).

`line-pixel-height` [Function]

This function returns the height in pixels of the line at point in the selected window. The value includes the line spacing of the line (see Section 41.11 [Line Height], page 1166).

`string-glyph-split` *string* [Function]

When character compositions are in effect, sequence of characters can be composed for display to form *grapheme clusters*, for example to display accented characters, or ligatures, or Emoji, or when complex text shaping requires that for some scripts.

When that happens, characters no longer map in a simple way to display columns, and display layout decisions with such strings, such as truncating too wide strings, can be a complex job. This function helps in performing such jobs: it splits up its argument *string* into a list of substrings, where each substring produces a single grapheme cluster that should be displayed as a unit. Lisp programs can then use this list to construct visually-valid substrings of *string* which will look correctly on display, or compute the width of any substring of *string* by adding the width of its constituents in the returned list, etc.

For instance, if you want to display a string without the first glyph, you can say:

```
(apply #'insert (cdr (string-glyph-split string)))
```

When a buffer is displayed with line numbers (see Section "Display Custom" in *The GNU Emacs Manual*), it is sometimes useful to know the width taken for displaying the line numbers. The following function is for Lisp programs which need this information for layout calculations.

`line-number-display-width` **&optional** *pixelwise* [Function]

This function returns the width used for displaying the line numbers in the selected window. If the optional argument *pixelwise* is the symbol `columns`, the return value is a float number of the frame's canonical columns; if *pixelwise* is `t` or any other non-`nil` value, the value is an integer and is measured in pixels. If *pixelwise* is omitted or `nil`, the value is the integer number of columns of the font defined for the `line-number` face, and doesn't include the 2 columns used to pad the numbers on display. If line numbers are not displayed in the selected window, the value is zero regardless of the value of *pixelwise*. Use `with-selected-window` (see Section 29.3 [Selecting Windows], page 698) if you need this information about another window.

41.11 Line Height

The total height of each display line consists of the height of the contents of the line, plus optional additional vertical line spacing above or below the display line.

The height of the line contents is the maximum height of any character or image on that display line, including the final newline if there is one. (A display line that is continued doesn't include a final newline.) That is the default line height, if you do nothing to specify a greater height. (In the most common case, this equals the height of the corresponding frame's default font, see Section 30.3.2 [Frame Font], page 799.)

There are several ways to explicitly specify a larger line height, either by specifying an absolute height for the display line, or by specifying vertical space. However, no matter what you specify, the actual line height can never be less than the default.

A newline can have a `line-height` text or overlay property that controls the total height of the display line ending in that newline. The property value can be one of several forms:

`t`

If the property value is `t`, the newline character has no effect on the displayed height of the line—the visible contents alone determine the height. The `line-spacing` property of the newline, described below, is also ignored in this case. This is useful for tiling small images (or image slices) without adding blank areas between the images.

`(height total)`
: If the property value is a list of the form shown, that adds extra space *below* the display line. First Emacs uses *height* as a height spec to control extra space *above* the line; then it adds enough space *below* the line to bring the total line height up to *total*. In this case, any value of `line-spacing` property for the newline is ignored.

Any other kind of property value is a height spec, which translates into a number—the specified line height. There are several ways to write a height spec; here's how each of them translates into a number:

integer
: If the height spec is a positive integer, the height value is that integer.

float
: If the height spec is a float, *float*, the numeric height value is *float* times the frame's default line height.

`(face . ratio)`
: If the height spec is a cons of the format shown, the numeric height is *ratio* times the height of face *face*. *ratio* can be any type of number, or `nil` which means a ratio of 1. If *face* is `t`, it refers to the current face.

`(nil . ratio)`
: If the height spec is a cons of the format shown, the numeric height is *ratio* times the height of the contents of the line.

Thus, any valid height spec determines the height in pixels, one way or another. If the line contents' height is less than that, Emacs adds extra vertical space above the line to achieve the specified total height.

If you don't specify the `line-height` property, the line's height consists of the contents' height plus the line spacing. There are several ways to specify the line spacing for different parts of Emacs text.

On graphical terminals, you can specify the line spacing for all lines in a frame, using the `line-spacing` frame parameter (see Section 30.4.3.4 [Layout Parameters], page 812). However, if the default value of `line-spacing` is non-`nil`, it overrides the frame's `line-spacing` parameter. An integer specifies the number of pixels put below lines. A floating-point number specifies the spacing relative to the frame's default line height.

You can specify the line spacing for all lines in a buffer via the buffer-local `line-spacing` variable. An integer specifies the number of pixels put below lines. A floating-point number specifies the spacing relative to the default frame line height. This overrides line spacings specified for the frame.

Finally, a newline can have a `line-spacing` text or overlay property that can enlarge the default frame line spacing and the buffer local `line-spacing` variable: if its value is larger than the buffer or frame defaults, that larger value is used instead, for the display line ending in that newline (unless the newline also has the `line-height` property whose value is one of the special values which cause `line-spacing` to be ignored, see above).

One way or another, these mechanisms specify a Lisp value for the spacing of each line. The value is a height spec, and it translates into a Lisp value as described above. However, in this case the numeric height value specifies the line spacing, rather than the line height.

On text terminals, the line spacing cannot be altered.

41.12 Faces

A *face* is a collection of graphical attributes for displaying text: font, foreground color, background color, optional underlining, etc. Faces control how Emacs displays text in buffers, as well as other parts of the frame such as the mode line.

One way to represent a face is as a property list of attributes, like `(:foreground "red" :weight bold)`. Such a list is called an *anonymous face*. For example, you can assign an anonymous face as the value of the `face` text property, and Emacs will display the underlying text with the specified attributes. See Section 33.19.4 [Special Properties], page 927.

More commonly, a face is referred to via a *face name*: a Lisp symbol associated with a set of face attributes[1]. Named faces are defined using the `defface` macro (see Section 41.12.2 [Defining Faces], page 1172). Emacs comes with several standard named faces (see Section 41.12.8 [Basic Faces], page 1182).

Some parts of Emacs require named faces (e.g., the functions documented in Section 41.12.3 [Attribute Functions], page 1175). Unless otherwise stated, we will use the term *face* to refer only to named faces.

`facep` *object* [Function]
: This function returns a non-`nil` value if *object* is a named face: a Lisp symbol or string which serves as a face name. Otherwise, it returns `nil`.

41.12.1 Face Attributes

Face attributes determine the visual appearance of a face. The following table lists all the face attributes, their possible values, and their effects.

Apart from the values given below, each face attribute can have the value `unspecified`. This special value means that the face doesn't specify that attribute directly. An `unspecified` attribute tells Emacs to refer instead to a parent face (see the description `:inherit` attribute below); or, failing that, to an underlying face (see Section 41.12.4 [Displaying Faces], page 1179). (However, `unspecified` is not a valid value in `defface`.)

A face attribute can also have the value `reset`. This special value stands for the value of the corresponding attribute of the `default` face.

The `default` face must explicitly specify all attributes, and cannot use the special value `reset`.

Some of these attributes are meaningful only on certain kinds of displays. If your display cannot handle a certain attribute, the attribute is ignored.

`:family` Font family name (a string). See Section "Fonts" in *The GNU Emacs Manual*, for more information about font families. The function `font-family-list` (see below) returns a list of available family names.

`:foundry` The name of the *font foundry* for the font family specified by the `:family` attribute (a string). See Section "Fonts" in *The GNU Emacs Manual*.

`:width` Relative character width. This should be one of the symbols `ultra-condensed`, `extra-condensed`, `condensed`, `semi-condensed`, `normal`, `regular`, `medium`, `semi-expanded`, `expanded`, `extra-expanded`, or `ultra-expanded`.

[1] For backward compatibility, you can also use a string to specify a face name; that is equivalent to a Lisp symbol with the same name.

`:height` The height of the font. In the simplest case, this is an integer in units of 1/10 point.

The value can also be floating point or a function, which specifies the height relative to an *underlying face* (see Section 41.12.4 [Displaying Faces], page 1179). A floating-point value specifies the amount by which to scale the height of the underlying face. A function value is called with one argument, the height of the underlying face, and returns the height of the new face. If the function is passed an integer argument, it must return an integer.

The height of the default face must be specified using an integer; floating point and function values are not allowed.

`:weight` Font weight—one of the symbols (from densest to faintest) `ultra-bold`, `extra-bold`, `bold`, `semi-bold`, `normal`, `semi-light`, `light`, `extra-light`, or `ultra-light`. On text terminals which support variable-brightness text, any weight greater than normal is displayed as extra bright, and any weight less than normal is displayed as half-bright.

`:slant` Font slant—one of the symbols `italic`, `oblique`, `normal`, `reverse-italic`, or `reverse-oblique`. On text terminals that support variable-brightness text, slanted text is displayed as half-bright.

`:foreground`
Foreground color, a string. The value can be a system-defined color name, or a hexadecimal color specification. See Section 30.23 [Color Names], page 848. On black-and-white displays, certain shades of gray are implemented by stipple patterns.

`:distant-foreground`
Alternative foreground color, a string. This is like `:foreground` but the color is only used as a foreground when the background color is near to the foreground that would have been used. This is useful for example when marking text (i.e., the region face). If the text has a foreground that is visible with the region face, that foreground is used. If the foreground is near the region face background, `:distant-foreground` is used instead so the text is readable.

`:background`
Background color, a string. The value can be a system-defined color name, or a hexadecimal color specification. See Section 30.23 [Color Names], page 848.

`:underline`
Whether or not characters should be underlined, and in what way. The possible values of the `:underline` attribute are:

`nil` Don't underline.

`t` Underline with the foreground color of the face.

color Underline in color *color*, a string specifying a color.

`(:color` *color* `:style` *style* `:position` *position*`)`
color is either a string, or the symbol `foreground-color`, meaning the foreground color of the face. Omitting the attribute `:color`

means to use the foreground color of the face. *style* should be a symbol `line` or `wave`, meaning to use a straight or wavy line. Omitting the attribute `:style` means to use a straight line. *position*, if non-`nil`, means to display the underline at the descent of the text, instead of at the baseline level. If it is a number, then it specifies the amount of pixels above the descent to display the underline.

`:overline`
: Whether or not characters should be overlined, and in what color. If the value is `t`, overlining uses the foreground color of the face. If the value is a string, overlining uses that color. The value `nil` means do not overline.

`:strike-through`
: Whether or not characters should be strike-through, and in what color. The value is used like that of `:overline`.

`:box`
: Whether or not a box should be drawn around characters, its color, the width of the box lines, and 3D appearance. Here are the possible values of the `:box` attribute, and what they mean:

  `nil`
  : Don't draw a box.

  `t`
  : Draw a box with lines of width 1, in the foreground color.

  color
  : Draw a box with lines of width 1, in color *color*.

  `(:line-width (`*vwidth* `.` *hwidth*`) :color` *color* `:style` *style*`)`
  : You can explicitly specify all aspects of the box with a plist on this form. Any element in this plist can be omitted.

    The values of *vwidth* and *hwidth* specify respectively the width of the vertical and horizontal lines to draw; they default to (1 . 1). A negative horizontal or vertical width $-n$ means to draw a line of width n that occupies the space of the underlying text, thus avoiding any increase in the character height or width. For simplification the width could be specified with only a single number n instead of a list, such case is equivalent to `((abs` *n*`) .` *n*`)`.

    The value of *color* specifies the color to draw with. The default is the background color of the face for 3D boxes and `flat-button`, and the foreground color of the face for other boxes.

    The value of *style* specifies whether to draw a 3D box. If it is `released-button`, the box looks like a 3D button that is not being pressed. If it is `pressed-button`, the box looks like a 3D button that is being pressed. If it is `nil`, `flat-button` or omitted, a plain 2D box is used.

  If you use the `:box` face attribute on strings displayed instead of buffer text via the `display` text property, special considerations might apply if the surrounding buffer text also has the `:box` face attribute. See Section 41.16.1 [Replacing Specs], page 1204.

`:inverse-video`
: Whether or not characters should be displayed in inverse video. The value should be `t` (yes) or `nil` (no).

`:stipple`
: The background stipple, a bitmap.

  The value can be a string; that should be the name of a file containing external-format X bitmap data. The file is found in the directories listed in the variable `x-bitmap-file-path`.

  Alternatively, the value can specify the bitmap directly, with a list of the form `(`*`width height data`*`)`. Here, *width* and *height* specify the size in pixels, and *data* is a string containing the raw bits of the bitmap, row by row. Each row occupies (*width*+7)/8 consecutive bytes in the string (which should be a unibyte string for best results). This means that each row always occupies at least one whole byte.

  If the value is `nil`, that means use no stipple pattern.

  Normally you do not need to set the stipple attribute, because it is used automatically to handle certain shades of gray.

`:font`
: The font used to display the face. Its value should be a font object or a fontset. If it is a font object, it specifies the font to be used by the face for displaying ASCII characters. See Section 41.12.12 [Low-Level Font], page 1188, for information about font objects, font specs, and font entities. See Section 41.12.11 [Fontsets], page 1186, for information about fontsets.

  When specifying this attribute using `set-face-attribute` or `set-face-font` (see Section 41.12.3 [Attribute Functions], page 1175), you may also supply a font spec, a font entity, or a string. Emacs converts such values to an appropriate font object, and stores that font object as the actual attribute value. If you specify a string, the contents of the string should be a font name (see Section "Fonts" in *The GNU Emacs Manual*); if the font name is an XLFD containing wildcards, Emacs chooses the first font matching those wildcards. Specifying this attribute also changes the values of the `:family`, `:foundry`, `:width`, `:height`, `:weight`, and `:slant` attributes.

`:inherit`
: The name of a face from which to inherit attributes, or a list of face names. Attributes from inherited faces are merged into the face like an underlying face would be, with higher priority than underlying faces (see Section 41.12.4 [Displaying Faces], page 1179). If the face to inherit from is `unspecified`, it is treated the same as `nil`, since Emacs never merges `:inherit` attributes. If a list of faces is used, attributes from faces earlier in the list override those from later faces.

`:extend`
: Whether or not this face will be extended beyond end of line and will affect the display of the empty space between the end of line and the edge of the window. The value should be `t` to display the empty space between end of line and edge of the window using this face, or `nil` to not use this face for the space between the end of the line and the edge of the window. When Emacs merges several faces for displaying the empty space beyond end of line, only those faces with `:extend` non-`nil` will be merged. By default, only a small

number of faces, notably, `region`, have this attribute set. This attribute is different from the others in that when a theme doesn't specify an explicit value for a face, the value from the original face definition by `defface` is inherited (see Section 41.12.2 [Defining Faces], page 1172).

Some modes, like `hl-line-mode`, use a face with an `:extend` property to mark the entire current line. Note, however, that Emacs will always allow you to move point after the final character in a buffer, and if the buffer ends with a newline character, point can be placed on what is seemingly a line at the end of the buffer—but Emacs can't highlight that "line", because it doesn't really exist.

`font-family-list` **&optional** *frame* [Function]
This function returns a list of available font family names. The optional argument *frame* specifies the frame on which the text is to be displayed; if it is `nil`, the selected frame is used.

`underline-minimum-offset` [User Option]
This variable specifies the minimum distance between the baseline and the underline, in pixels, when displaying underlined text.

`x-bitmap-file-path` [User Option]
This variable specifies a list of directories for searching for bitmap files, for the `:stipple` attribute.

`bitmap-spec-p` *object* [Function]
This returns `t` if *object* is a valid bitmap specification, suitable for use with `:stipple` (see above). It returns `nil` otherwise.

41.12.2 Defining Faces

The usual way to define a face is through the `defface` macro. This macro associates a face name (a symbol) with a default *face spec*. A face spec is a construct which specifies what attributes a face should have on any given terminal; for example, a face spec might specify one foreground color on high-color terminals, and a different foreground color on low-color terminals.

People are sometimes tempted to create a variable whose value is a face name. In the vast majority of cases, this is not necessary; the usual procedure is to define a face with `defface`, and then use its name directly.

Note that once you have defined a face (usually with `defface`), you cannot later undefine this face safely, except by restarting Emacs.

`defface` *face spec doc* [*keyword value*]. . . [Macro]
This macro declares *face* as a named face whose default face spec is given by *spec*. You should not quote the symbol *face*, and it should not end in '`-face`' (that would be redundant). The argument *doc* is a documentation string for the face. The additional *keyword* arguments have the same meanings as in `defgroup` and `defcustom` (see Section 15.1 [Common Keywords], page 278).

If *face* already has a default face spec, this macro does nothing.

The default face spec determines *face*'s appearance when no customizations are in effect (see Chapter 15 [Customization], page 278). If *face* has already been customized (via Custom themes or via customizations read from the init file), its appearance is determined by the custom face spec(s), which override the default face spec *spec*. However, if the customizations are subsequently removed, the appearance of *face* will again be determined by its default face spec.

As an exception, if you evaluate a `defface` form with `C-M-x` (`eval-defun`) or with `C-x C-e` (`eval-last-sexp`) in Emacs Lisp mode, a special feature of these commands overrides any custom face specs on the face, causing the face to reflect exactly what the `defface` says.

The *spec* argument is a *face spec*, which states how the face should appear on different kinds of terminals. It should be an alist whose elements each have the form

```
(display . plist)
```

display specifies a class of terminals (see below). *plist* is a property list of face attributes and their values, specifying how the face appears on such terminals. For backward compatibility, you can also write an element as `(display plist)`.

The *display* part of an element of *spec* determines which terminals the element matches. If more than one element of *spec* matches a given terminal, the first element that matches is the one used for that terminal. There are three possibilities for *display*:

`default` This element of *spec* doesn't match any terminal; instead, it specifies defaults that apply to all terminals. This element, if used, must be the first element of *spec*. Each of the following elements can override any or all of these defaults.

`t` This element of *spec* matches all terminals. Therefore, any subsequent elements of *spec* are never used. Normally `t` is used in the last (or only) element of *spec*.

a list If *display* is a list, each element should have the form `(characteristic value...)`. Here *characteristic* specifies a way of classifying terminals, and the *value*s are possible classifications which *display* should apply to. Here are the possible values of *characteristic*:

`type` The kind of window system the terminal uses—either `graphic` (any graphics-capable display), `x`, `pc` (for the MS-DOS console), `w32` (for MS Windows 9X/NT/2K/XP), `haiku` (for Haiku), `pgtk` (for pure GTK), or `tty` (a non-graphics-capable display). See Section 41.25 [Window Systems], page 1253.

`class` What kinds of colors the terminal supports—either `color`, `grayscale`, or `mono`.

`background`
The kind of background—either `light` or `dark`.

`min-colors`
: An integer that represents the minimum number of colors the terminal should support. This matches a terminal if its `display-color-cells` value is at least the specified integer.

`supports`
: Whether or not the terminal can display the face attributes given in *value*... (see Section 41.12.1 [Face Attributes], page 1168). See [Display Face Attribute Testing], page 852, for more information on exactly how this testing is done.

If an element of *display* specifies more than one *value* for a given *characteristic*, any of those values is acceptable. If *display* has more than one element, each element should specify a different *characteristic*; then *each* characteristic of the terminal must match one of the *value*s specified for it in *display*.

For example, here's the definition of the standard face `highlight`:

```
(defface highlight
  '((((class color) (min-colors 88) (background light))
     :background "darkseagreen2")
    (((class color) (min-colors 88) (background dark))
     :background "darkolivegreen")
    (((class color) (min-colors 16) (background light))
     :background "darkseagreen2")
    (((class color) (min-colors 16) (background dark))
     :background "darkolivegreen")
    (((class color) (min-colors 8))
     :background "green" :foreground "black")
    (t :inverse-video t))
  "Basic face for highlighting."
  :group 'basic-faces)
```

Internally, Emacs stores each face's default spec in its `face-defface-spec` symbol property (see Section 9.4 [Symbol Properties], page 138). The `saved-face` property stores any face spec saved by the user using the customization buffer; the `customized-face` property stores the face spec customized for the current session, but not saved; and the `theme-face` property stores an alist associating the active customization settings and Custom themes with the face specs for that face. The face's documentation string is stored in the `face-documentation` property.

Normally, a face is declared just once, using `defface`, and any further changes to its appearance are applied using the Customize framework (e.g., via the Customize user interface or via the `custom-set-faces` function; see Section 15.5 [Applying Customizations], page 295), or by face remapping (see Section 41.12.5 [Face Remapping], page 1179). In the rare event that you need to change a face spec directly from Lisp, you can use the `face-spec-set` function.

`face-spec-set` *face spec* **&optional** *spec-type* [Function]
: This function applies *spec* as a face spec for `face`. *spec* should be a face spec, as described in the above documentation for `defface`.

This function also defines *face* as a valid face name if it is not already one, and (re)calculates its attributes on existing frames.

The optional argument *spec-type* determines which spec to set. If it is omitted or `nil` or `face-override-spec`, this function sets the *override spec*, which overrides face specs on *face* of all the other types mentioned below. This is useful when calling this function outside of Custom code. If *spec-type* is `customized-face` or `saved-face`, this function sets the customized spec or the saved custom spec, respectively. If it is `face-defface-spec`, this function sets the default face spec (the same one set by `defface`). If it is `reset`, this function clears out all customization specs and override specs from *face* (in this case, the value of *spec* is ignored). The effect of any other value of *spec-type* on the face specs is reserved for internal use, but the function will still define *face* itself and recalculate its attributes, as described above.

41.12.3 Face Attribute Functions

This section describes functions for directly accessing and modifying the attributes of a named face.

`face-attribute` *face attribute* **&optional** *frame inherit* [Function]

This function returns the value of the *attribute* attribute for *face* on *frame*. See Section 41.12.1 [Face Attributes], page 1168, for the supported attributes.

If *frame* is omitted or `nil`, that means the selected frame (see Section 30.10 [Input Focus], page 827). If *frame* is `t`, this function returns the value of the specified attribute for newly-created frames, i.e. the value of the attribute before applying the face spec in the face's `defface` definition (see Section 41.12.2 [Defining Faces], page 1172) or the spec set by `face-spec-set`. This default value of *attribute* is normally `unspecified`, unless you have specified some other value using `set-face-attribute`; see below.

If *inherit* is `nil`, only attributes directly defined by *face* are considered, so the return value may be `unspecified`, or a relative value. If *inherit* is non-`nil`, *face*'s definition of *attribute* is merged with the faces specified by its `:inherit` attribute; however the return value may still be `unspecified` or relative. If *inherit* is a face or a list of faces, then the result is further merged with that face (or faces), until it becomes specified and absolute.

To ensure that the return value is always specified and absolute, use a value of `default` for *inherit*; this will resolve any unspecified or relative values by merging with the `default` face (which is always completely specified).

For example,

```
(face-attribute 'bold :weight)
     ⇒ bold
```

`face-attribute-relative-p` *attribute value* [Function]

This function returns non-`nil` if *value*, when used as the value of the face attribute *attribute*, is relative. This means it would modify, rather than completely override, any value that comes from a subsequent face in the face list or that is inherited from another face.

`unspecified` is a relative value for all attributes. For `:height`, floating point and function values are also relative.

For example:

```
(face-attribute-relative-p :height 2.0)
     ⇒ t
```

`face-all-attributes` *face* **&optional** *frame* [Function]
This function returns an alist of attributes of *face*. The elements of the result are name-value pairs of the form `(attr-name . attr-value)`. Optional argument *frame* specifies the frame whose definition of *face* to return; if omitted or `nil`, the returned value describes the default attributes of *face* for newly created frames, i.e. the values these attributes have before applying the face spec in the face's `defface` definition or the spec set by `face-spec-set`. These default values of the attributes are normally `unspecified`, unless you have specified some other value using `set-face-attribute`; see below.

`merge-face-attribute` *attribute value1 value2* [Function]
If *value1* is a relative value for the face attribute *attribute*, returns it merged with the underlying value *value2*; otherwise, if *value1* is an absolute value for the face attribute *attribute*, returns *value1* unchanged.

Normally, Emacs uses the face specs of each face to automatically calculate its attributes on each frame (see Section 41.12.2 [Defining Faces], page 1172). The function `set-face-attribute` can override this calculation by directly assigning attributes to a face, either on a specific frame or for all frames. This function is mostly intended for internal usage.

`set-face-attribute` *face frame* **&rest** *arguments* [Function]
This function sets one or more attributes of *face* for *frame*. The attributes specified in this way override the face spec(s) belonging to *face*. See Section 41.12.1 [Face Attributes], page 1168, for the supported attributes.

The extra arguments *arguments* specify the attributes to set, and the values for them. They should consist of alternating attribute names (such as `:family` or `:underline`) and values. Thus,

```
(set-face-attribute 'foo nil :weight 'bold :slant 'italic)
```

sets the attribute `:weight` to `bold` and the attribute `:slant` to `italic`.

If *frame* is `t`, this function sets the default attributes for newly created frames; they will effectively override the attribute values specified by `defface`. If *frame* is `nil`, this function sets the attributes for all existing frames, as well as for newly created frames.

To *reset* the value of an attribute, that is, to indicate that the face doesn't by itself specify a value for the attribute, use the special value `unspecified` (*not* `nil`!) for the attribute, and set the *frame* argument to `t`, in addition to the call with *frame* set to `nil`. This is because the default attributes for newly created frames are merged with the face's spec in `defface` when a new frame is created, and so having `unspecified` in the default attributes for new frames will be unable to override `defface`; the special call to this function as described above will arrange for `defface` to be overridden.

Note that the attribute-value pairs are evaluated in the order they are specified, with the exception of the `:family` and `:foundry` attributes, which are evaluated first. This means that if a certain attribute is specified more than once, only the last value will be used. It also means that in some cases a different order of attributes will produce different results. For example, when `:weight` is placed before `:font`, the weight value is applied to the current font of the face, and might be rounded to the closest available weight of that font, whereas when `:font` is placed before `:weight` the weight value is applied to the specified font.

The following commands and functions mostly provide compatibility with old versions of Emacs. They work by calling `set-face-attribute`. Values of `t` and `nil` (or omitted) for their *frame* argument are handled just like `set-face-attribute` and `face-attribute`. The commands read their arguments using the minibuffer, if called interactively.

`set-face-foreground` *face color* **&optional** *frame* [Command]
`set-face-background` *face color* **&optional** *frame* [Command]
These set the `:foreground` attribute (or `:background` attribute, respectively) of *face* to *color*.

`set-face-stipple` *face pattern* **&optional** *frame* [Command]
This sets the `:stipple` attribute of *face* to *pattern*.

`set-face-font` *face font* **&optional** *frame* [Command]
Change the font-related attributes of *face* to those of *font* (a string or a font object). See [face-font-attribute], page 1171, for the supported formats of the *font* argument. This function sets the attribute `:font` of the face, and indirectly also the `:family`, `:foundry`, `:width`, `:height`, `:weight`, and `:slant` attributes, as defined by the font. If *frame* is non-`nil`, only change the attributes on the specified frame.

`set-face-bold` *face bold-p* **&optional** *frame* [Function]
This sets the `:weight` attribute of *face* to *normal* if *bold-p* is `nil`, and to *bold* otherwise.

`set-face-italic` *face italic-p* **&optional** *frame* [Function]
This sets the `:slant` attribute of *face* to *normal* if *italic-p* is `nil`, and to *italic* otherwise.

`set-face-underline` *face underline* **&optional** *frame* [Command]
This sets the `:underline` attribute of *face* to *underline*.

`set-face-inverse-video` *face inverse-video-p* **&optional** *frame* [Command]
This sets the `:inverse-video` attribute of *face* to *inverse-video-p*.

`invert-face` *face* **&optional** *frame* [Command]
This swaps the foreground and background colors of face *face*.

`set-face-extend` *face extend* **&optional** *frame* [Command]
This sets the `:extend` attribute of *face* to *extend*.

The following functions examine the attributes of a face. They mostly provide compatibility with old versions of Emacs. If you don't specify *frame*, they refer to the selected frame; `t` refers to the default data for new frames. They return `unspecified` if the face doesn't define any value for that attribute. If *inherit* is `nil`, only an attribute directly defined by the face is returned. If *inherit* is non-`nil`, any faces specified by its `:inherit` attribute are considered as well, and if *inherit* is a face or a list of faces, then they are also considered, until a specified attribute is found. To ensure that the return value is always specified, use a value of `default` for *inherit*.

`face-font` *face* **&optional** *frame character* [Function]
: This function returns the name of the font used by the specified *face*.

  If the optional argument *frame* is specified, it returns the name of the font of *face* for that frame; *frame* defaults to the selected frame if it is `nil` or omitted. If *frame* is `t`, the function reports on the font defaults for *face* to be used for new frames.

  By default, the returned font is for displaying ASCII characters, but if *frame* is anything but `t`, and the optional third argument *character* is supplied, the function returns the font name used by *face* for that character.

`face-foreground` *face* **&optional** *frame inherit* [Function]
`face-background` *face* **&optional** *frame inherit* [Function]
: These functions return the foreground color (or background color, respectively) of face *face*, as a string. If the color is unspecified, they return `nil`.

`face-stipple` *face* **&optional** *frame inherit* [Function]
: This function returns the name of the background stipple pattern of face *face*, or `nil` if it doesn't have one.

`face-bold-p` *face* **&optional** *frame inherit* [Function]
: This function returns a non-`nil` value if the `:weight` attribute of *face* is bolder than normal (i.e., one of `semi-bold`, `bold`, `extra-bold`, or `ultra-bold`). Otherwise, it returns `nil`.

`face-italic-p` *face* **&optional** *frame inherit* [Function]
: This function returns a non-`nil` value if the `:slant` attribute of *face* is `italic` or `oblique`, and `nil` otherwise.

`face-underline-p` *face* **&optional** *frame inherit* [Function]
: This function returns non-`nil` if face *face* specifies a non-`nil` `:underline` attribute.

`face-inverse-video-p` *face* **&optional** *frame inherit* [Function]
: This function returns non-`nil` if face *face* specifies a non-`nil` `:inverse-video` attribute.

`face-extend-p` *face* **&optional** *frame inherit* [Function]
: This function returns non-`nil` if face *face* specifies a non-`nil` `:extend` attribute. The *inherit* argument is passed to `face-attribute`.

41.12.4 Displaying Faces

When Emacs displays a given piece of text, the visual appearance of the text may be determined by faces drawn from different sources. If these various sources together specify more than one face for a particular character, Emacs merges the attributes of the various faces. Here is the order in which Emacs merges the faces, from highest to lowest priority:

- If the text consists of a special glyph, the glyph can specify a particular face. See Section 41.23.4 [Glyphs], page 1250.
- If the text lies within an active region, Emacs highlights it using the `region` face. See Section "Standard Faces" in *The GNU Emacs Manual*.
- If the text lies within an overlay with a non-`nil` `face` property, Emacs applies the face(s) specified by that property. If the overlay has a `mouse-face` property and the mouse is near enough to the overlay, Emacs applies the face or face attributes specified by the `mouse-face` property instead. See Section 41.9.2 [Overlay Properties], page 1157.

  When multiple overlays cover the same character, an overlay with higher priority overrides those with lower priority. See Section 41.9 [Overlays], page 1154.
- If the text contains a `face` or `mouse-face` property, Emacs applies the specified faces and face attributes. See Section 33.19.4 [Special Properties], page 927. (This is how Font Lock mode faces are applied. See Section 24.6 [Font Lock Mode], page 564.)
- If the text lies within the mode line of the selected window, Emacs applies the `mode-line` face. For the mode line of a non-selected window, Emacs applies the `mode-line-inactive` face. For a header line, Emacs applies the `header-line` face. For a tab line, Emacs applies the `tab-line` face.
- If the text comes from an overlay string via `before-string` or `after-string` properties (see Section 41.9.2 [Overlay Properties], page 1157), or from a display string (see Section 41.16.4 [Other Display Specs], page 1207), and the string doesn't contain a `face` or `mouse-face` property, or these properties leave some face attributes undefined, but the buffer text affected by the overlay/display property does define a face or those attributes, Emacs applies the face attributes of the "underlying" buffer text. Note that this is so even if the overlay or display string is displayed in the display margins (see Section 41.16.5 [Display Margins], page 1209).
- If any given attribute has not been specified during the preceding steps, Emacs applies the attribute of the `default` face.

At each stage, if a face has a valid `:inherit` attribute, Emacs treats any attribute with an `unspecified` value as having the corresponding value drawn from the parent face(s). see Section 41.12.1 [Face Attributes], page 1168. Note that the parent face(s) may also leave the attribute unspecified; in that case, the attribute remains unspecified at the next level of face merging.

41.12.5 Face Remapping

The variable `face-remapping-alist` is used for buffer-local or global changes in the appearance of a face. For instance, it is used to implement the `text-scale-adjust` command (see Section "Text Scale" in *The GNU Emacs Manual*).

`face-remapping-alist` [Variable]

The value of this variable is an alist whose elements have the form `(face . remapping)`. This causes Emacs to display any text having the face *face* with *remapping*, rather than the ordinary definition of *face*.

remapping may be any face spec suitable for a `face` text property: either a face (i.e., a face name or a property list of attribute/value pairs), or a list of faces. For details, see the description of the `face` text property in Section 33.19.4 [Special Properties], page 927. *remapping* serves as the complete specification for the remapped face—it replaces the normal definition of *face*, instead of modifying it.

If `face-remapping-alist` is buffer-local, its local value takes effect only within that buffer. If `face-remapping-alist` includes faces applicable only to certain windows, by using the `(:filtered (:window param val) spec)`, that face takes effect only in windows that match the filter conditions (see Section 33.19.4 [Special Properties], page 927). To turn off face filtering temporarily, bind `face-filters-always-match` to a non-`nil` value, then all face filters will match any window.

Note: face remapping is non-recursive. If *remapping* references the same face name *face*, either directly or via the `:inherit` attribute of some other face in *remapping*, that reference uses the normal definition of *face*. For instance, if the `mode-line` face is remapped using this entry in `face-remapping-alist`:

```
(mode-line italic mode-line)
```

then the new definition of the `mode-line` face inherits from the `italic` face, and the *normal* (non-remapped) definition of `mode-line` face.

The following functions implement a higher-level interface to `face-remapping-alist`. Most Lisp code should use these functions instead of setting `face-remapping-alist` directly, to avoid trampling on remappings applied elsewhere. These functions are intended for buffer-local remappings, so they all make `face-remapping-alist` buffer-local as a side-effect. They manage `face-remapping-alist` entries of the form

```
(face relative-spec-1 relative-spec-2 ... base-spec)
```

where, as explained above, each of the *relative-spec-N* and *base-spec* is either a face name, or a property list of attribute/value pairs. Each of the *relative remapping* entries, *relative-spec-N*, is managed by the `face-remap-add-relative` and `face-remap-remove-relative` functions; these are intended for simple modifications like changing the text size. The *base remapping* entry, *base-spec*, has the lowest priority and is managed by the `face-remap-set-base` and `face-remap-reset-base` functions; it is intended for major modes to remap faces in the buffers they control.

`face-remap-add-relative` *face* **&rest** *specs* [Function]

This function adds *specs* as relative remappings for face *face* in the current buffer. *specs* should be a list where each element is either a face name, or a property list of attribute/value pairs.

The return value is a Lisp object that serves as a cookie; you can pass this object as an argument to `face-remap-remove-relative` if you need to remove the remapping later.

```
;; Remap the 'escape-glyph' face into a combination
```

```
;; of the 'highlight' and 'italic' faces:
(face-remap-add-relative 'escape-glyph 'highlight 'italic)

;; Increase the size of the 'default' face by 50%:
(face-remap-add-relative 'default :height 1.5)
```

Note that buffer-local face remapping does not work reliably for parent faces of basic faces (see Section 41.12.8 [Basic Faces], page 1182). (These are the faces that are used in mode lines, header lines, and other basic decorations of windows and frames.) For instance, `mode-line-inactive` inherits from `mode-line`, but remapping `mode-line` won't normally have the desired effect on `mode-line-inactive`, especially if done locally for some buffers. Instead you have to remap `mode-line-inactive` directly.

`face-remap-remove-relative` *cookie* [Function]
This function removes a relative remapping previously added by `face-remap-add-relative`. *cookie* should be the Lisp object returned by `face-remap-add-relative` when the remapping was added.

`face-remap-set-base` *face* **&rest** *specs* [Function]
This function sets the base remapping of *face* in the current buffer to *specs*. If *specs* is empty, the default base remapping is restored, similar to calling `face-remap-reset-base` (see below); note that this is different from *specs* containing a single value `nil`, which has the opposite result (the global definition of *face* is ignored).

This overwrites the default *base-spec*, which inherits the global face definition, so it is up to the caller to add such inheritance if so desired.

`face-remap-reset-base` *face* [Function]
This function sets the base remapping of *face* to its default value, which inherits from *face*'s global definition.

41.12.6 Functions for Working with Faces

Here are additional functions for creating and working with faces.

`face-list` [Function]
This function returns a list of all defined face names.

`face-id` *face* [Function]
This function returns the *face number* of face *face*. This is a number that uniquely identifies a face at low levels within Emacs. It is seldom necessary to refer to a face by its face number. However, functions that manipulate glyphs, such as `make-glyph-code` and `glyph-face` (see Section 41.23.4 [Glyphs], page 1250) access the face numbers internally. Note that the face number is stored as the value of the `face` property of the face symbol, so we recommend not to set that property of a face to any value of your own.

`face-documentation` *face* [Function]
This function returns the documentation string of face *face*, or `nil` if none was specified for it.

`face-equal` *face1 face2* **&optional** *frame* [Function]
This returns `t` if the faces *face1* and *face2* have the same attributes for display.

`face-differs-from-default-p` *face* **&optional** *frame* [Function]
This returns non-`nil` if the face *face* displays differently from the default face.

A *face alias* provides an equivalent name for a face. You can define a face alias by giving the alias symbol the `face-alias` property, with a value of the target face name. The following example makes `modeline` an alias for the `mode-line` face.

```
(put 'modeline 'face-alias 'mode-line)
```

`define-obsolete-face-alias` *obsolete-face current-face when* [Macro]
This macro defines `obsolete-face` as an alias for *current-face*, and also marks it as obsolete, indicating that it may be removed in future. *when* should be a string indicating when `obsolete-face` was made obsolete (usually a version number string).

41.12.7 Automatic Face Assignment

This hook is used for automatically assigning faces to text in the buffer. It is part of the implementation of Jit-Lock mode, used by Font-Lock.

`fontification-functions` [Variable]
This variable holds a list of functions that are called by Emacs redisplay as needed, just before doing redisplay. They are called even when Font Lock Mode isn't enabled. When Font Lock Mode is enabled, this variable usually holds just one function, `jit-lock-function`.

The functions are called in the order listed, with one argument, a buffer position *pos*. Collectively they should attempt to assign faces to the text in the current buffer starting at *pos*.

The functions should record the faces they assign by setting the `face` property. They should also add a non-`nil` `fontified` property to all the text they have assigned faces to. That property tells redisplay that faces have been assigned to that text already.

It is probably a good idea for the functions to do nothing if the character after *pos* already has a non-`nil` `fontified` property, but this is not required. If one function overrides the assignments made by a previous one, the properties after the last function finishes are the ones that really matter.

For efficiency, we recommend writing these functions so that they usually assign faces to around 400 to 600 characters at each call.

Note that, when the buffer text includes very long lines, these functions are called as if they were in a `with-restriction` form (see Section 31.4 [Narrowing], page 867), with a `long-line-optimizations-in-fontification-functions` label and with the buffer narrowed to a portion around *pos*.

41.12.8 Basic Faces

If your Emacs Lisp program needs to assign some faces to text, it is often a good idea to use certain existing faces or inherit from them, rather than defining entirely new faces. This way, if other users have customized those existing faces to give Emacs a certain look, your program will fit in without additional customization.

Some of the *basic faces* defined in Emacs are listed below. In addition to these, you might want to make use of the Font Lock faces for syntactic highlighting, if highlighting is not already handled by Font Lock mode, or if some Font Lock faces are not in use. See Section 24.6.7 [Faces for Font Lock], page 574.

`default` The default face, whose attributes are all specified. All other faces implicitly inherit from it: any unspecified attribute defaults to the attribute on this face (see Section 41.12.1 [Face Attributes], page 1168).

`mode-line-active`
`mode-line-inactive`
`header-line`
`tab-line` Basic faces used for the mode line, header line, and tab line.

`tool-bar`
`tab-bar`
`fringe`
`scroll-bar`
`window-divider`
`border`
`child-frame-border`
Basic faces used for the corresponding decorations of GUI frames.

`cursor` The basic face used for the text cursor.

`mouse` The basic face used for displaying mouse-sensitive text when the mouse pointer is on that text.

`bold`
`italic`
`bold-italic`
`underline`
`fixed-pitch`
`fixed-pitch-serif`
`variable-pitch`
These have the attributes indicated by their names (e.g., `bold` has a bold `:weight` attribute), with all other attributes unspecified (and so given by `default`).

`shadow` For dimmed-out text. For example, it is used for the ignored part of a filename in the minibuffer (see Section "Minibuffers for File Names" in *The GNU Emacs Manual*).

`link`
`link-visited`
For clickable text buttons that send the user to a different buffer or location.

`highlight`
For stretches of text that should temporarily stand out. For example, it is commonly assigned to the `mouse-face` property for cursor highlighting (see Section 33.19.4 [Special Properties], page 927).

`match`
`isearch`
`lazy-highlight`
: For text matching (respectively) permanent search matches, interactive search matches, and lazy highlighting other matches than the current interactive one.

`error`
`warning`
`success`
: For text concerning errors, warnings, or successes. For example, these are used for messages in `*Compilation*` buffers.

41.12.9 Font Selection

Before Emacs can draw a character on a graphical display, it must select a *font* for that character[2]. See Section "Fonts" in *The GNU Emacs Manual*. Normally, Emacs automatically chooses a font based on the faces assigned to that character—specifically, the face attributes `:family`, `:weight`, `:slant`, and `:width` (see Section 41.12.1 [Face Attributes], page 1168). The choice of font also depends on the character to be displayed; some fonts can only display a limited set of characters. If no available font exactly fits the requirements, Emacs looks for the *closest matching font*. The variables in this section control how Emacs makes this selection.

`face-font-family-alternatives` [User Option]
: If a given family is specified but does not exist, this variable specifies alternative font families to try. Each element should have this form:

  `(`*`family alternate-families...`*`)`

  If *family* is specified but not available, Emacs will try the other families given in *alternate-families*, one by one, until it finds a family that does exist.

`face-font-selection-order` [User Option]
: If there is no font that exactly matches all desired face attributes (`:width`, `:height`, `:weight`, and `:slant`), this variable specifies the order in which these attributes should be considered when selecting the closest matching font. The value should be a list containing those four attribute symbols, in order of decreasing importance. The default is `(:width :height :weight :slant)`.

  Font selection first finds the best available matches for the first attribute in the list; then, among the fonts which are best in that way, it searches for the best matches in the second attribute, and so on.

  The attributes `:weight` and `:width` have symbolic values in a range centered around `normal`. Matches that are more extreme (farther from `normal`) are somewhat preferred to matches that are less extreme (closer to `normal`); this is designed to ensure that non-normal faces contrast with normal ones, whenever possible.

  One example of a case where this variable makes a difference is when the default font has no italic equivalent. With the default ordering, the `italic` face will use a non-italic font that is similar to the default one. But if you put `:slant` before `:height`, the `italic` face will use an italic font, even if its height is not quite right.

[2] In this context, the term *font* has nothing to do with Font Lock (see Section 24.6 [Font Lock Mode], page 564).

`face-font-registry-alternatives` [User Option]
This variable lets you specify alternative font registries to try, if a given registry is specified and doesn't exist. Each element should have this form:

```
(registry alternate-registries...)
```

If *registry* is specified but not available, Emacs will try the other registries given in *alternate-registries*, one by one, until it finds a registry that does exist.

Emacs can make use of scalable fonts, but by default it does not use them.

`scalable-fonts-allowed` [User Option]
This variable controls which scalable fonts to use. A value of `nil`, the default, means do not use scalable fonts. `t` means to use any scalable font that seems appropriate for the text.

Otherwise, the value must be a list of regular expressions. Then a scalable font is enabled for use if its name matches any regular expression in the list. For example,

```
(setq scalable-fonts-allowed '("iso10646-1$"))
```

allows the use of scalable fonts with registry `iso10646-1`.

`face-font-rescale-alist` [Variable]
This variable specifies scaling for certain faces. Its value should be a list of elements of the form

```
(fontname-regexp . scale-factor)
```

If *fontname-regexp* matches the font name that is about to be used, this says to choose a larger similar font according to the factor *scale-factor*. You would use this feature to normalize the font size if certain fonts are bigger or smaller than their nominal heights and widths would suggest.

41.12.10 Looking Up Fonts

`x-list-fonts` *name* **&optional** *reference-face frame maximum width* [Function]
This function returns a list of available font names that match *name*. *name* should be a string containing a font name in either the Fontconfig, GTK+, or XLFD format (see Section "Fonts" in *The GNU Emacs Manual*). Within an XLFD string, wildcard characters may be used: the '`*`' character matches any substring, and the '`?`' character matches any single character. Case is ignored when matching font names.

If the optional arguments *reference-face* and *frame* are specified, the returned list includes only fonts that are the same size as *reference-face* (a face name) currently is on the frame *frame*.

The optional argument *maximum* sets a limit on how many fonts to return. If it is non-`nil`, then the return value is truncated after the first *maximum* matching fonts. Specifying a small value for *maximum* can make this function much faster, in cases where many fonts match the pattern.

The optional argument *width* specifies a desired font width. If it is non-`nil`, the function only returns those fonts whose characters are (on average) *width* times as wide as *reference-face*.

`x-family-fonts` **&optional** *family frame* [Function]

This function returns a list describing the available fonts for family *family* on *frame*. If *family* is omitted or `nil`, this list applies to all families, and therefore, it contains all available fonts. Otherwise, *family* must be a string; it may contain the wildcards '`?`' and '`*`'.

The list describes the display that *frame* is on; if *frame* is omitted or `nil`, it applies to the selected frame's display (see Section 30.10 [Input Focus], page 827).

Each element in the list is a vector of the following form:

```
[family width point-size weight slant
 fixed-p full registry-and-encoding]
```

The first five elements correspond to face attributes; if you specify these attributes for a face, it will use this font.

The last three elements give additional information about the font. *fixed-p* is non-`nil` if the font is fixed-pitch. *full* is the full name of the font, and *registry-and-encoding* is a string giving the registry and encoding of the font.

41.12.11 Fontsets

A *fontset* is a list of fonts, each assigned to a range of character codes. An individual font cannot display the whole range of characters that Emacs supports, but a fontset can. Fontsets have names, just as fonts do, and you can use a fontset name in place of a font name when you specify the font for a frame or a face. Here is information about defining a fontset under Lisp program control.

`create-fontset-from-fontset-spec` *fontset-spec* **&optional** *style-variant-p noerror* [Function]

This function defines a new fontset according to the specification string *fontset-spec*. The string should have this format:

```
fontpattern, [charset:font]...
```

Whitespace characters before and after the commas are ignored.

The first part of the string, *fontpattern*, should have the form of a standard X font name, except that the last two fields should be '`fontset-alias`'.

The new fontset has two names, one long and one short. The long name is *fontpattern* in its entirety. The short name is '`fontset-alias`'. You can refer to the fontset by either name. If a fontset with the same name already exists, an error is signaled, unless *noerror* is non-`nil`, in which case this function does nothing.

If optional argument *style-variant-p* is non-`nil`, that says to create bold, italic and bold-italic variants of the fontset as well. These variant fontsets do not have a short name, only a long one, which is made by altering *fontpattern* to indicate the bold and/or italic status.

The specification string also says which fonts to use in the fontset. See below for the details.

The construct '*charset*:*font*' specifies which font to use (in this fontset) for one particular character set. Here, *charset* is the name of a character set, and *font* is the font to use

for that character set. You can use this construct any number of times in the specification string.

For the remaining character sets, those that you don't specify explicitly, Emacs chooses a font based on *fontpattern*: it replaces '`fontset-alias`' with a value that names one character set. For the ASCII character set, '`fontset-alias`' is replaced with '`ISO8859-1`'.

In addition, when several consecutive fields are wildcards, Emacs collapses them into a single wildcard. This is to prevent use of auto-scaled fonts. Fonts made by scaling larger fonts are not usable for editing, and scaling a smaller font is not useful because it is better to use the smaller font in its own size, which Emacs does.

Thus if *fontpattern* is this,

```
-*-fixed-medium-r-normal-*-24-*-*-*-*-*-fontset-24
```

the font specification for ASCII characters would be this:

```
-*-fixed-medium-r-normal-*-24-*-ISO8859-1
```

and the font specification for Chinese GB2312 characters would be this:

```
-*-fixed-medium-r-normal-*-24-*-gb2312*-*
```

You may not have any Chinese font matching the above font specification. Most X distributions include only Chinese fonts that have '`song ti`' or '`fangsong ti`' in the *family* field. In such a case, '`Fontset-n`' can be specified as below:

```
Emacs.Fontset-0: -*-fixed-medium-r-normal-*-24-*-*-*-*-*-fontset-24,\
        chinese-gb2312:-*-*-medium-r-normal-*-24-*-gb2312*-*
```

Then, the font specifications for all but Chinese GB2312 characters have '`fixed`' in the *family* field, and the font specification for Chinese GB2312 characters has a wild card '`*`' in the *family* field.

`set-fontset-font` *fontset characters font-spec* **&optional** *frame add* [Function]

This function modifies the existing *fontset* to use the font specified by *font-spec* for displaying the specified *characters*.

If *fontset* is `nil`, this function modifies the fontset of the selected frame or that of *frame* if *frame* is not `nil`.

If *fontset* is `t`, this function modifies the default fontset, whose short name as a string is '`fontset-default`'.

The *characters* argument can be a single character which should be displayed using *font-spec*. It can also be a cons cell `(from . to)`, where *from* and *to* are characters. In that case, use *font-spec* for all the characters in the range *from* and *to* (inclusive).

characters may be a charset symbol (see Section 34.7 [Character Sets], page 976). In that case, use *font-spec* for all the characters in the charset.

characters may be a script symbol (see Section 34.6 [Character Properties], page 971). In that case, use *font-spec* for all the characters belonging to the script.

characters may be `nil`, which means to use *font-spec* for any character in *fontset* for which no font-spec is specified.

font-spec may be a font-spec object created by the function `font-spec` (see Section 41.12.12 [Low-Level Font], page 1188).

font-spec may be a cons cell (`family . registry`), where *family* is a family name of a font (possibly including a foundry name at the head), and *registry* is a registry name of a font (possibly including an encoding name at the tail).

font-spec may be a font name, a string.

font-spec may be `nil`, which explicitly specifies that there's no font for the specified *characters*. This is useful, for example, to avoid expensive system-wide search for fonts for characters that have no glyphs, like those from the Unicode Private Use Area (PUA).

The optional argument *add*, if non-`nil`, specifies how to add *font-spec* to the font specifications previously set for *characters*. If it is `prepend`, *font-spec* is prepended to the existing specs. If it is `append`, *font-spec* is appended. By default, *font-spec* overwrites the previously set font specs.

For instance, this changes the default fontset to use a font whose family name is '`Kochi Gothic`' for all characters belonging to the charset `japanese-jisx0208`:

```
(set-fontset-font t 'japanese-jisx0208
                  (font-spec :family "Kochi Gothic"))
```

Note that this function should generally be called from the user's init files, and more generally before any of *characters* were displayed in the current Emacs session. That's because for some scripts, Emacs caches the way they are displayed, and the cached information includes the font used for them – once these characters are displayed once, the cached font will continue to be used regardless of changes in the fontsets.

`char-displayable-p` *char* [Function]

This function returns non-`nil` if Emacs ought to be able to display *char*. Or more precisely, if the selected frame's fontset has a font to display the character set that *char* belongs to.

Fontsets can specify a font on a per-character basis; when the fontset does that, this function's value may not be accurate.

This function may return non-`nil` even when there is no font available, since it also checks whether the coding system for the text terminal can encode the character (see Section 34.10.8 [Terminal I/O Encoding], page 993).

41.12.12 Low-Level Font Representation

Normally, it is not necessary to manipulate fonts directly. In case you need to do so, this section explains how.

In Emacs Lisp, fonts are represented using three different Lisp object types: *font objects*, *font specs*, and *font entities*.

`fontp` *object* **&optional** *type* [Function]

Return `t` if *object* is a font object, font spec, or font entity. Otherwise, return `nil`.

The optional argument *type*, if non-`nil`, determines the exact type of Lisp object to check for. In that case, *type* should be one of `font-object`, `font-spec`, or `font-entity`.

A font object is a Lisp object that represents a font that Emacs has *opened*. Font objects cannot be modified in Lisp, but they can be inspected.

`font-at` *position* **&optional** *window string* [Function]
: Return the font object that is being used to display the character at position *position* in the window *window*. If *window* is `nil`, it defaults to the selected window. If *string* is `nil`, *position* specifies a position in the current buffer; otherwise, *string* should be a string, and *position* specifies a position in that string.

A font spec is a Lisp object that contains a set of specifications that can be used to find a font. More than one font may match the specifications in a font spec.

`font-spec` **&rest** *arguments* [Function]
: Return a new font spec using the specifications in *arguments*, which should come in `property-value` pairs. The possible specifications are as follows:

`:name`
: The font name (a string), in either XLFD, Fontconfig, or GTK+ format. See Section "Fonts" in *The GNU Emacs Manual*.

`:family`
`:foundry`
`:weight`
`:slant`
`:width`
: These have the same meanings as the face attributes of the same name. See Section 41.12.1 [Face Attributes], page 1168. `:family` and `:foundry` are strings, while the other three are symbols. As example values, `:slant` may be `italic`, `:weight` may be `bold` and `:width` may be `normal`.

`:size`
: The font size—either a non-negative integer that specifies the pixel size, or a floating-point number that specifies the point size.

`:adstyle`
: Additional typographic style information for the font, such as '`sans`'. The value should be a string or a symbol.

`:registry`
: The charset registry and encoding of the font, such as '`iso8859-1`'. The value should be a string or a symbol.

`:dpi`
: The resolution in dots per inch for which the font is designed. The value must be a non-negative number.

`:spacing`
: The spacing of the font: proportional, dual, mono, or charcell. The value should be either an integer (0 for proportional, 90 for dual, 100 for mono, 110 for charcell) or a one-letter symbol (one of `P`, `D`, `M`, or `C`).

`:avgwidth`
: The average width of the font in 1/10 pixel units. The value should be a non-negative number.

`:script`
: The script that the font must support (a symbol).

`:lang`
: The language that the font should support. The value should be a symbol whose name is a two-letter ISO-639 language name. On X, the value is matched against the "Additional Style" field of the XLFD name of a font, if it is non-empty. On MS-Windows, fonts matching the spec are required to support codepages needed for the language. Currently, only a small set of CJK languages is supported with this property: '`ja`', '`ko`', and '`zh`'.

`:otf` The font must be an OpenType font that supports these OpenType features, provided Emacs is compiled with a library, such as '`libotf`' on GNU/Linux, that supports complex text layout for scripts which need that. The value must be a list of the form

```
(script-tag langsys-tag gsub gpos)
```

where *script-tag* is the OpenType script tag symbol; *langsys-tag* is the OpenType language system tag symbol, or `nil` to use the default language system; `gsub` is a list of OpenType GSUB feature tag symbols, or `nil` if none is required; and `gpos` is a list of OpenType GPOS feature tag symbols, or `nil` if none is required. If `gsub` or `gpos` is a list, a `nil` element in that list means that the font must not match any of the remaining tag symbols. The `gpos` element may be omitted. For the list of OpenType script, language, and feature tags, see the list of registered OTF tags (`https://docs.microsoft.com/en-us/typography/opentype/spec/ttoreg`).

`:type` The symbol that specifies the *font backend* used to draw the characters. The possible values depend on the platform and on how Emacs was configured at build time. Typical values include `ftcrhb` and `xfthb` on X, `harfbuzz` on MS-Windows, `ns` on GNUstep, etc. It can also be `nil` if left unspecified, typically in a font-spec.

`font-put` *font-spec property value* [Function]
Set the font property *property* in the font-spec *font-spec* to *value*. The *property* can any of the ones described above.

A font entity is a reference to a font that need not be open. Its properties are intermediate between a font object and a font spec: like a font object, and unlike a font spec, it refers to a single, specific font. Unlike a font object, creating a font entity does not load the contents of that font into computer memory. Emacs may open multiple font objects of different sizes from a single font entity referring to a scalable font.

`find-font` *font-spec* **&optional** *frame* [Function]
This function returns a font entity that best matches the font spec *font-spec* on frame *frame*. If *frame* is `nil`, it defaults to the selected frame.

`list-fonts` *font-spec* **&optional** *frame num prefer* [Function]
This function returns a list of all font entities that match the font spec *font-spec*.

The optional argument *frame*, if non-`nil`, specifies the frame on which the fonts are to be displayed. The optional argument *num*, if non-`nil`, should be an integer that specifies the maximum length of the returned list. The optional argument *prefer*, if non-`nil`, should be another font spec, which is used to control the order of the returned list; the returned font entities are sorted in order of decreasing closeness to that font spec.

If you call `set-face-attribute` and pass a font spec, font entity, or font name string as the value of the `:font` attribute, Emacs opens the best matching font that is available for display. It then stores the corresponding font object as the actual value of the `:font` attribute for that face.

The following functions can be used to obtain information about a font. For these functions, the *font* argument can be a font object, a font entity, or a font spec.

`font-get` *font property* [Function]

This function returns the value of the font property *property* for *font*. The *property* can any of the ones that `font-spec` supports.

If *font* is a font spec and the font spec does not specify *property*, the return value is `nil`. If *font* is a font object or font entity, the value for the *:script* property may be a list of scripts supported by the font, and the value of the `:otf` property is a cons of the form `(gsub . gpos)`, where *gsub* and *gpos* are lists representing OpenType features supported by the font, of the form

```
((script-tag (langsys-tag feature...) ...) ...)
```

where *script-tag*, *langsys-tag*, and *feature* are symbols representing OpenType layout tags.

If *font* is a font object, the special property `:combining-capability` is non-`nil` if the font backend of *font* supports rendering of combining characters for non-OpenType fonts.

`font-face-attributes` *font* **&optional** *frame* [Function]

This function returns a list of face attributes corresponding to *font*. The optional argument *frame* specifies the frame on which the font is to be displayed. If it is `nil`, the selected frame is used. The return value has the form

```
(:family family :height height :weight weight
   :slant slant :width width)
```

where the values of *family*, *height*, *weight*, *slant*, and *width* are face attribute values. Some of these key-attribute pairs may be omitted from the list if they are not specified by *font*.

`font-xlfd-name` *font* **&optional** *fold-wildcards* [Function]

This function returns the XLFD (X Logical Font Descriptor), a string, matching *font*. See Section "Fonts" in *The GNU Emacs Manual*, for information about XLFDs. If the name is too long for an XLFD (which can contain at most 255 characters), the function returns `nil`.

If the optional argument *fold-wildcards* is non-`nil`, consecutive wildcards in the XLFD are folded into one.

The following two functions return important information about a font.

`font-info` *name* **&optional** *frame* [Function]

This function returns information about a font specified by its *name*, a string, as it is used on *frame*. If *frame* is omitted or `nil`, it defaults to the selected frame.

The value returned by the function is a vector of the form `[opened-name full-name size height baseline-offset relative-compose default-ascent max-width ascent descent space-width average-width filename capability]`. Here's the description of each components of this vector:

opened-name

The name used to open the font, a string.

full-name The full name of the font, a string.

size The pixel size of the font.

height The height of the font in pixels.

baseline-offset
The offset in pixels from the ASCII baseline, positive upward.

relative-compose
default-ascent
Numbers controlling how to compose characters.

max-width
The maximum advance width of the font.

ascent
descent The ascent and descent of this font. The sum of these two numbers should be equal to the value of *height* above.

space-width
The width, in pixels, of the font's space character.

average-width
The average width of the font characters. If this is zero, Emacs uses the value of *space-width* instead, when it calculates text layout on display.

filename The file name of the font as a string. This can be `nil` if the font back-end does not provide a way to find out the font's file name.

capability A list whose first element is a symbol representing the font type, one of `x`, `opentype`, `truetype`, `type1`, `pcf`, or `bdf`. For OpenType fonts, the list includes 2 additional elements describing the GSUB and GPOS features supported by the font. Each of these elements is a list of the form `((`*`script`* `(`*`langsys feature ...`*`) ...) ...)`, where *script* is a symbol representing an OpenType script tag, *langsys* is a symbol representing an OpenType langsys tag (or `nil`, which stands for the default langsys), and each *feature* is a symbol representing an OpenType feature tag.

`query-font` *font-object* [Function]
This function returns information about a *font-object*. (This is in contrast to `font-info`, which takes the font name, a string, as its argument.)

The value returned by the function is a vector of the form `[`*`name filename pixel-size max-width ascent descent space-width average-width capability`*`]`. Here's the description of each components of this vector:

name The font name, a string.

filename The file name of the font as a string. This can be `nil` if the font back-end does not provide a way to find out the font's file name.

pixel-size The pixel size of the font used to open the font.

max-width
The maximum advance width of the font.

ascent
descent The ascent and descent of this font. The sum of these two numbers gives the font height.

space-width
The width, in pixels, of the font's space character.

average-width
The average width of the font characters. If this is zero, Emacs uses the value of *space-width* instead, when it calculates text layout on display.

capability A list whose first element is a symbol representing the font type, one of `x`, `opentype`, `truetype`, `type1`, `pcf`, or `bdf`. For OpenType fonts, the list includes 2 additional elements describing the GSUB and GPOS features supported by the font. Each of these elements is a list of the form `((`*`script`* `(`*`langsys feature`* `...) ...) ...)`, where *script* is a symbol representing an OpenType script tag, *langsys* is a symbol representing an OpenType langsys tag (or `nil`, which stands for the default langsys), and each *feature* is a symbol representing an OpenType feature tag.

The following four functions return size information about fonts used by various faces, allowing various layout considerations in Lisp programs. These functions take face remapping into consideration, returning information about the remapped face, if the face in question was remapped. See Section 41.12.5 [Face Remapping], page 1179.

`default-font-width` [Function]
This function returns the average width in pixels of the font used by the current buffer's default face, as that face is defined for the selected frame.

`default-font-height` [Function]
This function returns the height in pixels of the font used by the current buffer's default face, as that face is defined for the selected frame.

`window-font-width` **&optional** *window face* [Function]
This function returns the average width in pixels for the font used by *face* in *window*. The specified *window* must be a live window. If `nil` or omitted, *window* defaults to the selected window, and *face* defaults to the default face in *window*.

`window-font-height` **&optional** *window face* [Function]
This function returns the height in pixels for the font used by *face* in *window*. The specified *window* must be a live window. If `nil` or omitted, *window* defaults to the selected window, and *face* defaults to the default face in *window*.

41.13 Fringes

On graphical displays, Emacs draws *fringes* next to each window: thin vertical strips down the sides which can display bitmaps indicating truncation, continuation, horizontal scrolling, and so on.

41.13.1 Fringe Size and Position

The following buffer-local variables control the position and width of fringes in windows showing that buffer.

`fringes-outside-margins` [Variable]
: The fringes normally appear between the display margins and the window text. If the value is non-`nil`, they appear outside the display margins. See Section 41.16.5 [Display Margins], page 1209.

`left-fringe-width` [Variable]
: This variable, if non-`nil`, specifies the width of the left fringe in pixels. A value of `nil` means to use the left fringe width from the window's frame.

`right-fringe-width` [Variable]
: This variable, if non-`nil`, specifies the width of the right fringe in pixels. A value of `nil` means to use the right fringe width from the window's frame.

Any buffer which does not specify values for these variables uses the values specified by the `left-fringe` and `right-fringe` frame parameters (see Section 30.4.3.4 [Layout Parameters], page 812).

The above variables actually take effect via the function `set-window-buffer` (see Section 29.11 [Buffers and Windows], page 722), which calls `set-window-fringes` as a subroutine. If you change one of these variables, the fringe display is not updated in existing windows showing the buffer, unless you call `set-window-buffer` again in each affected window. You can also use `set-window-fringes` to control the fringe display in individual windows.

`set-window-fringes` *window left* **&optional** *right outside-margins persistent* [Function]
: This function sets the fringe widths of window *window*. If *window* is `nil`, the selected window is used.

  The argument *left* specifies the width in pixels of the left fringe, and likewise *right* for the right fringe. A value of `nil` for either one stands for the default width. If *outside-margins* is non-`nil`, that specifies that fringes should appear outside of the display margins.

  If *window* is not large enough to accommodate fringes of the desired width, this leaves the fringes of *window* unchanged.

  The values specified here may be later overridden by invoking `set-window-buffer` (see Section 29.11 [Buffers and Windows], page 722) on *window* with its *keep-margins* argument `nil` or omitted. However, if the optional fifth argument *persistent* is non-`nil` and the other arguments are processed successfully, the values specified here unconditionally survive subsequent invocations of `set-window-buffer`. This can be used to permanently turn off fringes in the minibuffer window, consult the description of `set-window-scroll-bars` for an example (see Section 41.14 [Scroll Bars], page 1199).

`window-fringes` **&optional** *window* [Function]
This function returns information about the fringes of a window *window*. If *window* is omitted or `nil`, the selected window is used. The value has the form (*left-width right-width outside-margins persistent*).

41.13.2 Fringe Indicators

Fringe indicators are tiny icons displayed in the window fringe to indicate truncated or continued lines, buffer boundaries, etc.

`indicate-empty-lines` [User Option]
When this is non-`nil`, Emacs displays a special glyph in the fringe of each empty line at the end of the buffer, on graphical displays. See Section 41.13 [Fringes], page 1193. This variable is automatically buffer-local in every buffer.

`indicate-buffer-boundaries` [User Option]
This buffer-local variable controls how the buffer boundaries and window scrolling are indicated in the window fringes.

Emacs can indicate the buffer boundaries—that is, the first and last line in the buffer—with angle icons when they appear on the screen. In addition, Emacs can display an up-arrow in the fringe to show that there is text above the screen, and a down-arrow to show there is text below the screen.

There are three kinds of basic values:

`nil` Don't display any of these fringe icons.

`left` Display the angle icons and arrows in the left fringe.

`right` Display the angle icons and arrows in the right fringe.

any non-alist
Display the angle icons in the left fringe and don't display the arrows.

Otherwise the value should be an alist that specifies which fringe indicators to display and where. Each element of the alist should have the form (*indicator* . *position*). Here, *indicator* is one of `top`, `bottom`, `up`, `down`, and `t` (which covers all the icons not yet specified), while *position* is one of `left`, `right` and `nil`.

For example, `((top . left) (t . right))` places the top angle bitmap in left fringe, and the bottom angle bitmap as well as both arrow bitmaps in right fringe. To show the angle bitmaps in the left fringe, and no arrow bitmaps, use `((top . left) (bottom . left))`.

`fringe-indicator-alist` [Variable]
This buffer-local variable specifies the mapping from logical fringe indicators to the actual bitmaps displayed in the window fringes. The value is an alist of elements (*indicator* . *bitmaps*), where *indicator* specifies a logical indicator type and *bitmaps* specifies the fringe bitmaps to use for that indicator.

Each *indicator* should be one of the following symbols:

`truncation`, `continuation`.
Used for truncation and continuation lines.

`up`, `down`, `top`, `bottom`, `top-bottom`
: Used when `indicate-buffer-boundaries` is non-`nil`: `up` and `down` indicate a buffer boundary lying above or below the window edge; `top` and `bottom` indicate the topmost and bottommost buffer text line; and `top-bottom` indicates where there is just one line of text in the buffer.

`empty-line`
: Used to indicate empty lines after the buffer end when `indicate-empty-lines` is non-`nil`.

`overlay-arrow`
: Used for overlay arrows (see Section 41.13.6 [Overlay Arrow], page 1199).

Each *bitmaps* value may be a list of symbols (`(`*`left right`* `[`*`left1 right1`*`]`). The *left* and *right* symbols specify the bitmaps shown in the left and/or right fringe, for the specific indicator. *left1* and *right1* are specific to the `bottom` and `top-bottom` indicators, and are used to indicate that the last text line has no final newline. Alternatively, *bitmaps* may be a single symbol which is used in both left and right fringes.

See Section 41.13.4 [Fringe Bitmaps], page 1197, for a list of standard bitmap symbols and how to define your own. In addition, `nil` represents the empty bitmap (i.e., an indicator that is not shown).

When `fringe-indicator-alist` has a buffer-local value, and there is no bitmap defined for a logical indicator, or the bitmap is `t`, the corresponding value from the default value of `fringe-indicator-alist` is used.

41.13.3 Fringe Cursors

When a line is exactly as wide as the window, Emacs displays the cursor in the right fringe instead of using two lines. Different bitmaps are used to represent the cursor in the fringe depending on the current buffer's cursor type.

`overflow-newline-into-fringe` [User Option]
: If this is non-`nil`, lines exactly as wide as the window (not counting the final newline character) are not continued. Instead, when point is at the end of the line, the cursor appears in the right fringe.

`fringe-cursor-alist` [Variable]
: This variable specifies the mapping from logical cursor type to the actual fringe bitmaps displayed in the right fringe. The value is an alist where each element has the form (`(`*`cursor-type . bitmap`*`)`), which means to use the fringe bitmap *bitmap* to display cursors of type *cursor-type*.

  Each *cursor-type* should be one of `box`, `hollow`, `bar`, `hbar`, or `hollow-small`. The first four have the same meanings as in the `cursor-type` frame parameter (see Section 30.4.3.9 [Cursor Parameters], page 819). The `hollow-small` type is used instead of `hollow` when the normal `hollow-rectangle` bitmap is too tall to fit on a specific display line.

  Each *bitmap* should be a symbol specifying the fringe bitmap to be displayed for that logical cursor type. See the next subsection for details.

When `fringe-cursor-alist` has a buffer-local value, and there is no bitmap defined for a cursor type, the corresponding value from the default value of `fringes-indicator-alist` is used.

41.13.4 Fringe Bitmaps

The *fringe bitmaps* are the actual bitmaps which represent the logical fringe indicators for truncated or continued lines, buffer boundaries, overlay arrows, etc. Each bitmap is represented by a symbol. These symbols are referred to by the variables `fringe-indicator-alist` and `fringe-cursor-alist`, described in the previous subsections.

Lisp programs can also directly display a bitmap in the left or right fringe, by using a `display` property for one of the characters appearing in the line (see Section 41.16.4 [Other Display Specs], page 1207). Such a display specification has the form

```
(fringe bitmap [face])
```

fringe is either the symbol `left-fringe` or `right-fringe`. *bitmap* is a symbol identifying the bitmap to display. The optional *face* names a face whose foreground and background colors are to be used to display the bitmap, using the attributes of the `fringe` face for colors that *face* didn't specify. If *face* is omitted, that means to use the attributes of the `default` face for the colors which the `fringe` face didn't specify. For predictable results that don't depend on the attributes of the `default` and `fringe` faces, we recommend you never omit *face*, but always provide a specific face. In particular, if you want the bitmap to be always displayed in the `fringe` face, use `fringe` as *face*.

For instance, to display an arrow in the left fringe, using the `warning` face, you could say something like:

```
(overlay-put
 (make-overlay (point) (point))
 'before-string (propertize
                 "x" 'display
                 `(left-fringe right-arrow warning)))
```

Here is a list of the standard fringe bitmaps defined in Emacs, and how they are currently used in Emacs (via `fringe-indicator-alist` and `fringe-cursor-alist`):

`left-arrow`, `right-arrow`
: Used to indicate truncated lines.

`left-curly-arrow`, `right-curly-arrow`
: Used to indicate continued lines.

`right-triangle`, `left-triangle`
: The former is used by overlay arrows. The latter is unused.

`up-arrow`, `down-arrow`
`bottom-left-angle`, `bottom-right-angle`
`top-left-angle`, `top-right-angle`
`left-bracket`, `right-bracket`
`empty-line`
: Used to indicate buffer boundaries.

`filled-rectangle`, `hollow-rectangle`
`filled-square`, `hollow-square`
`vertical-bar`, `horizontal-bar`
: Used for different types of fringe cursors.

`exclamation-mark`, `question-mark`
`large-circle`
: Not used by core Emacs features.

The next subsection describes how to define your own fringe bitmaps.

`fringe-bitmaps-at-pos` **&optional** *pos window* [Function]
: This function returns the fringe bitmaps of the display line containing position *pos* in window *window*. The return value has the form `(left right ov)`, where *left* is the symbol for the fringe bitmap in the left fringe (or `nil` if no bitmap), *right* is similar for the right fringe, and *ov* is non-`nil` if there is an overlay arrow in the left fringe.

  The value is `nil` if *pos* is not visible in *window*. If *window* is `nil`, that stands for the selected window. If *pos* is `nil`, that stands for the value of point in *window*.

41.13.5 Customizing Fringe Bitmaps

`define-fringe-bitmap` *bitmap bits* **&optional** *height width align* [Function]
: This function defines the symbol *bitmap* as a new fringe bitmap, or replaces an existing bitmap with that name.

  The argument *bits* specifies the image to use. It should be either a string or a vector of integers, where each element (an integer) corresponds to one row of the bitmap. Each bit of an integer corresponds to one pixel of the bitmap, where the low bit corresponds to the rightmost pixel of the bitmap. (Note that this order of bits is opposite of the order in XBM images; see Section 41.17.3 [XBM Images], page 1216.)

  The height is normally the length of *bits*. However, you can specify a different height with non-`nil` *height*. The width is normally 8, but you can specify a different width with non-`nil` *width*. The width must be an integer between 1 and 16.

  The argument *align* specifies the positioning of the bitmap relative to the range of rows where it is used; the default is to center the bitmap. The allowed values are `top`, `center`, or `bottom`.

  The *align* argument may also be a list `(align periodic)` where *align* is interpreted as described above. If *periodic* is non-`nil`, it specifies that the rows in `bits` should be repeated enough times to reach the specified height.

`destroy-fringe-bitmap` *bitmap* [Function]
: This function destroys the fringe bitmap identified by *bitmap*. If *bitmap* identifies a standard fringe bitmap, it actually restores the standard definition of that bitmap, instead of eliminating it entirely.

`set-fringe-bitmap-face` *bitmap* **&optional** *face* [Function]
: This sets the face for the fringe bitmap *bitmap* to *face*. If *face* is `nil`, it selects the `fringe` face. The bitmap's face controls the color to draw it in.

  face is merged with the `fringe` face, so normally *face* should specify only the foreground color.

41.13.6 The Overlay Arrow

The *overlay arrow* is useful for directing the user's attention to a particular line in a buffer. For example, in the modes used for interface to debuggers, the overlay arrow indicates the line of code about to be executed. This feature has nothing to do with *overlays* (see Section 41.9 [Overlays], page 1154).

`overlay-arrow-string` [Variable]
: This variable holds the string to display to call attention to a particular line, or `nil` if the arrow feature is not in use. On a graphical display the contents of the string are ignored if the left fringe is shown; instead a glyph is displayed in the fringe area to the left of the display area.

`overlay-arrow-position` [Variable]
: This variable holds a marker that indicates where to display the overlay arrow. It should point at the beginning of a line. On a non-graphical display, or when the left fringe is not shown, the arrow text appears at the beginning of that line, overlaying any text that would otherwise appear. Since the arrow is usually short, and the line usually begins with indentation, normally nothing significant is overwritten.

  The overlay-arrow string is displayed in any given buffer if the value of `overlay-arrow-position` in that buffer points into that buffer. Thus, it is possible to display multiple overlay arrow strings by creating buffer-local bindings of `overlay-arrow-position`. However, it is usually cleaner to use `overlay-arrow-variable-list` to achieve this result.

You can do a similar job by creating an overlay with a `before-string` property. See Section 41.9.2 [Overlay Properties], page 1157.

You can define multiple overlay arrows via the variable `overlay-arrow-variable-list`.

`overlay-arrow-variable-list` [Variable]
: This variable's value is a list of variables, each of which specifies the position of an overlay arrow. The variable `overlay-arrow-position` has its normal meaning because it is on this list.

Each variable on this list can have properties `overlay-arrow-string` and `overlay-arrow-bitmap` that specify an overlay arrow string (for text terminals or graphical terminals without the left fringe shown) or fringe bitmap (for graphical terminals with a left fringe) to display at the corresponding overlay arrow position. If either property is not set, the default `overlay-arrow-string` or `overlay-arrow` fringe indicator is used.

41.14 Scroll Bars

Normally the frame parameter `vertical-scroll-bars` controls whether the windows in the frame have vertical scroll bars, and whether they are on the left or right. The frame parameter `scroll-bar-width` specifies how wide they are (`nil` meaning the default).

The frame parameter `horizontal-scroll-bars` controls whether the windows in the frame have horizontal scroll bars. The frame parameter `scroll-bar-height` specifies how high they are (`nil` meaning the default). See Section 30.4.3.4 [Layout Parameters], page 812.

Horizontal scroll bars are not available on all platforms. The function `horizontal-scroll-bars-available-p` which takes no argument returns non-`nil` if they are available on your system.

The following three functions take as argument a live frame which defaults to the selected one.

`frame-current-scroll-bars` **&optional** *frame* [Function]
: This function reports the scroll bar types for frame *frame*. The value is a cons cell (*vertical-type* . *horizontal-type*), where *vertical-type* is either `left`, `right`, or `nil` (which means no vertical scroll bar.) *horizontal-type* is either `bottom` or `nil` (which means no horizontal scroll bar).

`frame-scroll-bar-width` **&optional** *frame* [Function]
: This function returns the width of vertical scroll bars of *frame* in pixels.

`frame-scroll-bar-height` **&optional** *frame* [Function]
: This function returns the height of horizontal scroll bars of *frame* in pixels.

You can override the frame specific settings for individual windows by using the following function:

`set-window-scroll-bars` *window* **&optional** *width vertical-type height horizontal-type persistent* [Function]
: This function sets the width and/or height and the types of scroll bars for window *window*. If *window* is `nil`, the selected window is used.

  width specifies the width of the vertical scroll bar in pixels (`nil` means use the width specified for the frame). *vertical-type* specifies whether to have a vertical scroll bar and, if so, where. The possible values are `left`, `right`, `t`, which means to use the frame's default, and `nil` for no vertical scroll bar.

  height specifies the height of the horizontal scroll bar in pixels (`nil` means use the height specified for the frame). *horizontal-type* specifies whether to have a horizontal scroll bar. The possible values are `bottom`, `t`, which means to use the frame's default, and `nil` for no horizontal scroll bar. Note that for a mini window the value `t` has the same meaning as `nil`, namely to not show a horizontal scroll bar. You have to explicitly specify `bottom` in order to show a horizontal scroll bar in a mini window.

  If *window* is not large enough to accommodate a scroll bar of the desired dimension, this leaves the corresponding scroll bar unchanged.

  The values specified here may be later overridden by invoking `set-window-buffer` (see Section 29.11 [Buffers and Windows], page 722) on *window* with its *keep-margins* argument `nil` or omitted. However, if the optional fifth argument *persistent* is non-`nil` and the other arguments are processed successfully, the values specified here unconditionally survive subsequent invocations of `set-window-buffer`.

Using the *persistent* argument of `set-window-scroll-bars` and `set-window-fringes` (see Section 41.13.1 [Fringe Size/Pos], page 1194) you can reliably and permanently turn off scroll bars and/or fringes in any minibuffer window by adding the following snippet to your early init file (see Section 42.1.2 [Init File], page 1263).

```
(add-hook 'after-make-frame-functions
          (lambda (frame)
            (set-window-scroll-bars
             (minibuffer-window frame) 0 nil 0 nil t)
            (set-window-fringes
             (minibuffer-window frame) 0 0 nil t)))
```

The following four functions take as argument a live window which defaults to the selected one.

`window-scroll-bars` **&optional** *window* [Function]

This function returns a list of the form `(`*`width columns vertical-type height lines horizontal-type persistent`*`)`. The value *width* is the value that was specified for the width of the vertical scroll bar (which may be `nil`); *columns* is the (possibly rounded) number of columns that the vertical scroll bar actually occupies.

The value *height* is the value that was specified for the height of the horizontal scroll bar (which may be `nil`); *lines* is the (possibly rounded) number of lines that the horizontally scroll bar actually occupies.

The value of *persistent* is the value specified for *window* with the last successful invocation of `set-window-scroll-bars`, `nil` if there never was one.

`window-current-scroll-bars` **&optional** *window* [Function]

This function reports the scroll bar type for window *window*. The value is a cons cell `(`*`vertical-type . horizontal-type`*`)`. Unlike `window-scroll-bars`, this reports the scroll bar type actually used, once frame defaults and `scroll-bar-mode` are taken into account.

`window-scroll-bar-width` **&optional** *window* [Function]

This function returns the width in pixels of *window*'s vertical scrollbar.

`window-scroll-bar-height` **&optional** *window* [Function]

This function returns the height in pixels of *window*'s horizontal scrollbar.

If you do not specify a window's scroll bar settings via `set-window-scroll-bars`, the buffer-local variables `vertical-scroll-bar`, `horizontal-scroll-bar`, `scroll-bar-width` and `scroll-bar-height` in the buffer being displayed control the window's scroll bars. The function `set-window-buffer` examines these variables. If you change them in a buffer that is already visible in a window, you can make the window take note of the new values by calling `set-window-buffer` specifying the same buffer that is already displayed.

You can control the appearance of scroll bars for a particular buffer by setting the following variables which automatically become buffer-local when set.

`vertical-scroll-bar` [Variable]

This variable specifies the location of the vertical scroll bar. The possible values are `left`, `right`, `t`, which means to use the frame's default, and `nil` for no scroll bar.

`horizontal-scroll-bar` [Variable]

This variable specifies the location of the horizontal scroll bar. The possible values are `bottom`, `t`, which means to use the frame's default, and `nil` for no scroll bar.

`scroll-bar-width` [Variable]
: This variable specifies the width of the buffer's vertical scroll bars, measured in pixels. A value of `nil` means to use the value specified by the frame.

`scroll-bar-height` [Variable]
: This variable specifies the height of the buffer's horizontal scroll bar, measured in pixels. A value of `nil` means to use the value specified by the frame.

Finally you can toggle the display of scroll bars on all frames by customizing the variables `scroll-bar-mode` and `horizontal-scroll-bar-mode`.

`scroll-bar-mode` [User Option]
: This variable controls whether and where to put vertical scroll bars in all frames. The possible values are `nil` for no scroll bars, `left` to put scroll bars on the left and `right` to put scroll bars on the right.

`horizontal-scroll-bar-mode` [User Option]
: This variable controls whether to display horizontal scroll bars on all frames.

41.15 Window Dividers

Window dividers are bars drawn between a frame's windows. A right divider is drawn between a window and any adjacent windows on the right. Its width (thickness) is specified by the frame parameter `right-divider-width`. A bottom divider is drawn between a window and adjacent windows on the bottom or the echo area. Its width is specified by the frame parameter `bottom-divider-width`. In either case, specifying a width of zero means to not draw such dividers. See Section 30.4.3.4 [Layout Parameters], page 812.

Technically, a right divider belongs to the window on its left, which means that its width contributes to the total width of that window. A bottom divider belongs to the window above it, which means that its width contributes to the total height of that window. See Section 29.4 [Window Sizes], page 701. When a window has both, a right and a bottom divider, the bottom divider prevails. This means that a bottom divider is drawn over the full total width of its window while the right divider ends above the bottom divider.

Dividers can be dragged with the mouse and are therefore useful for adjusting the sizes of adjacent windows with the mouse. They also serve to visually set apart adjacent windows when no scroll bars or mode lines are present. The following three faces allow the customization of the appearance of dividers:

`window-divider`
: When a divider is less than three pixels wide, it is drawn solidly with the foreground of this face. For larger dividers this face is used for the inner part only, excluding the first and last pixel.

`window-divider-first-pixel`
: This is the face used for drawing the first pixel of a divider that is at least three pixels wide. To obtain a solid appearance, set this to the same value used for the `window-divider` face.

`window-divider-last-pixel`
: This is the face used for drawing the last pixel of a divider that is at least three pixels wide. To obtain a solid appearance, set this to the same value used for the `window-divider` face.

You can get the sizes of the dividers of a specific window with the following two functions.

`window-right-divider-width` **&optional** *window* [Function]
: Return the width (thickness) in pixels of *window*'s right divider. *window* must be a live window and defaults to the selected one. The return value is always zero for a rightmost window.

`window-bottom-divider-width` **&optional** *window* [Function]
: Return the width (thickness) in pixels of *window*'s bottom divider. *window* must be a live window and defaults to the selected one. The return value is zero for the minibuffer window or a bottommost window on a minibuffer-less frame.

41.16 The `display` Property

The `display` text property (or overlay property) is used to insert images into text, and to control other aspects of how text displays. Display specifications in the same `display` property value generally apply in parallel to the text they cover.

If several sources (overlays and/or a text property) specify values for the `display` property, only one of the values takes effect, following the rules of `get-char-property`. See Section 33.19.1 [Examining Properties], page 920.

The value of the `display` property should be a display specification, or a list or vector containing several display specifications.

`get-display-property` *position prop* **&optional** *object properties* [Function]
: This convenience function can be used to get a specific display property, no matter whether the `display` property is a vector, a list or a simple property. This is like `get-text-property` (see Section 33.19.1 [Examining Properties], page 920), but works on the `display` property only.

  position is the position in the buffer or string to examine, and *prop* is the `display` property to return. The optional *object* argument should be either a string or a buffer, and defaults to the current buffer. If the optional *properties* argument is non-`nil`, it should be a `display` property, and in that case, *position* and *object* are ignored. (This can be useful if you've already gotten the `display` property with `get-char-property`, for instance (see Section 33.19.1 [Examining Properties], page 920).

`add-display-text-property` *start end prop value* **&optional** *object* [Function]
: Add `display` property *prop* of *value* to the text from *start* to *end*.

  If any text in the region has a non-`nil` `display` property, those properties are retained. For instance:

```
(add-display-text-property 4 8 'height 2.0)
(add-display-text-property 2 12 'raise 0.5)
```

  After doing this, the region from 2 to 4 will have the `raise display` property, the region from 4 to 8 will have both the `raise` and `height display` properties, and finally the region from 8 to 12 will only have the `raise display` property.

If *object* is non-`nil`, it should be a string or a buffer. If `nil`, this defaults to the current buffer.

Some of the display specifications allow inclusion of Lisp forms, which are evaluated at display time. This could be unsafe in certain situations, e.g., when the display specification was generated by some external program/agent. Wrapping a display specification in a list that begins with the special symbol `disable-eval`, as in `(disable-eval` *`spec`*`)`, will disable evaluation of any Lisp in *spec*, while still supporting all the other display property features.

The rest of this section describes several kinds of display specifications and what they mean.

41.16.1 Display Specs That Replace The Text

Some kinds of display specifications specify something to display instead of the text that has the property. These are called *replacing* display specifications. Emacs does not allow the user to interactively move point into the middle of buffer text that is replaced in this way.

If a list of display specifications includes more than one replacing display specification, the first overrides the rest. Replacing display specifications make most other display specifications irrelevant, since those don't apply to the replacement.

For replacing display specifications, *the text that has the property* means all the consecutive characters that have the same Lisp object as their `display` property; these characters are replaced as a single unit. If two characters have different Lisp objects as their `display` properties (i.e., objects which are not `eq`), they are handled separately.

Here is an example which illustrates this point. A string serves as a replacing display specification, which replaces the text that has the property with the specified string (see Section 41.16.4 [Other Display Specs], page 1207). Consider the following function:

```
(defun foo ()
  (dotimes (i 5)
    (let ((string (concat "A"))
          (start (+ i i (point-min))))
      (put-text-property start (1+ start) 'display string)
      (put-text-property start (+ 2 start) 'display string))))
```

This function gives each of the first ten characters in the buffer a `display` property which is a string `"A"`, but they don't all get the same string object. The first two characters get the same string object, so they are replaced with one '`A`'; the fact that the display property was assigned in two separate calls to `put-text-property` is irrelevant. Similarly, the next two characters get a second string (`concat` creates a new string object), so they are replaced with one '`A`'; and so on. Thus, the ten characters appear as five A's.

Note: Using `:box` face attribute (see Section 41.12.1 [Face Attributes], page 1168) on a replacing `display` string that is adjacent to normal text with the same `:box` style can lead to display artifacts when moving the cursor across the text with this face attribute. These can be avoided by applying the `:box` attribute directly to the text being replaced, rather than (or in addition to) the `display` string itself. Here's an example:

```
;; Causes display artifacts when moving the cursor across text
(progn
  (put-text-property 1 2 'display (propertize "  [" 'face '(:box t)))
  (put-text-property 2 3 'face '(:box t))
  (put-text-property 3 4 'display (propertize "]  " 'face '(:box t))))
```

```
;; No display artifacts due to `:box'
(progn
  (add-text-properties 1 2 '(face (:box t) display "  ["))
  (put-text-property 2 3 'face '(:box t))
  (add-text-properties 3 4 '(face (:box t) display "]  ")))
```

41.16.2 Specified Spaces

To display a space of specified width and/or height, use a display specification of the form `(space . props)`, where *props* is a property list (a list of alternating properties and values). You can put this property on one or more consecutive characters; a space of the specified height and width is displayed in place of *all* of those characters. These are the properties you can use in *props* to specify the weight of the space:

`:width` *width*
: If *width* is a number, it specifies that the space width should be *width* times the normal character width. *width* can also be a *pixel width* specification (see Section 41.16.3 [Pixel Specification], page 1206).

`:relative-width` *factor*
: Specifies that the width of the stretch should be computed from the first character in the group of consecutive characters that have the same `display` property. The space width is the pixel width of that character, multiplied by *factor*. (On text-mode terminals, the "pixel width" of a character is usually 1, but it could be more for TABs and double-width CJK characters.)

`:align-to` *hpos*
: Specifies that the space should be wide enough to reach the column *hpos*. If *hpos* is a number, it is a column number, and is measured in units of the canonical character width (see Section 30.3.2 [Frame Font], page 799). *hpos* can also be a *pixel width* specification (see Section 41.16.3 [Pixel Specification], page 1206). When the current line is wider than the window, and is either displayed by one or more continuation lines, or is truncated and possibly scrolled horizontally (see Section 29.23 [Horizontal Scrolling], page 770), *hpos* is measured from the beginning of the logical line, not from the visual beginning of the screen line. This way, alignment produced by `:align-to` is consistent with functions that count columns, such as `current-column` and `move-to-column` (see Section 33.16 [Columns], page 912).

You should use one and only one of the above properties. You can also specify the height of the space, with these properties:

`:height` *height*
: Specifies the height of the space. If *height* is a number, it specifies that the space height should be *height* times the normal character height. The *height* may also be a *pixel height* specification (see Section 41.16.3 [Pixel Specification], page 1206).

`:relative-height` *factor*
: Specifies the height of the space, multiplying the ordinary height of the text having this display specification by *factor*.

`:ascent` *ascent*

> If the value of *ascent* is a non-negative number no greater than 100, it specifies that *ascent* percent of the height of the space should be considered as the ascent of the space—that is, the part above the baseline. The ascent may also be specified in pixel units with a *pixel ascent* specification (see Section 41.16.3 [Pixel Specification], page 1206).

Don't use both `:height` and `:relative-height` together.

The `:width` and `:align-to` properties are supported on non-graphic terminals, but the other space properties in this section are not.

Note that space properties are treated as paragraph separators for the purposes of reordering bidirectional text for display. See Section 41.27 [Bidirectional Display], page 1255, for the details.

41.16.3 Pixel Specification for Spaces

The value of the `:width`, `:align-to`, `:height`, and `:ascent` properties can be a special kind of expression that is evaluated during redisplay. The result of the evaluation is used as an absolute number of pixels.

The following expressions are supported:

```
expr ::= num | (num) | unit | elem | pos | image | xwidget | form
num  ::= integer | float | symbol
unit ::= in | mm | cm | width | height
elem ::= left-fringe | right-fringe | left-margin | right-margin
      |  scroll-bar | text
pos  ::= left | center | right
form ::= (num . expr) | (op expr ...)
op   ::= + | -
```

The form *num* specifies a fraction of the default frame font height or width. The form `(num)` specifies an absolute number of pixels. If *num* is a symbol, *symbol*, its buffer-local variable binding is used; that binding can be either a number or a cons cell of the forms shown above (including yet another cons cell whose `car` is a symbol that has a buffer-local binding).

The `in`, `mm`, and `cm` units specify the number of pixels per inch, millimeter, and centimeter, respectively. The `width` and `height` units correspond to the default width and height of the current face. An image specification of the form `(image . props)` (see Section 41.17.2 [Image Descriptors], page 1212) corresponds to the width or height of the specified image. Similarly, an xwidget specification of the form `(xwidget . props)` stands for the width or height of the specified xwidget. See Section 41.19 [Xwidgets], page 1233.

The elements `left-fringe`, `right-fringe`, `left-margin`, `right-margin`, `scroll-bar`, and `text` specify the width of the corresponding area of the window. When the window displays line numbers (see Section 41.10 [Size of Displayed Text], page 1161), the width of the `text` area is decreased by the screen space taken by the line-number display.

The `left`, `center`, and `right` positions can be used with `:align-to` to specify a position relative to the left edge, center, or right edge of the text area. When the window displays line numbers, and `:align-to` is used in display properties of buffer text (as opposed to header line, see below), the `left` and the `center` positions are offset to account for the screen space taken by the line-number display.

Any of the above window elements (except `text`) can also be used with `:align-to` to specify that the position is relative to the left edge of the given area. Once the base offset for a relative position has been set (by the first occurrence of one of these symbols), further occurrences of these symbols are interpreted as the width of the specified area. For example, to align to the center of the left-margin, use

```
:align-to (+ left-margin (0.5 . left-margin))
```

If no specific base offset is set for alignment, it is always relative to the left edge of the text area. For example, '`:align-to 0`' aligns with the first text column in the text area. When the window displays line numbers, the text is considered to start where the space used for line-number display ends.

A value of the form `(`*`num`* `.` *`expr`*`)` stands for the product of the values of *num* and *expr*. For example, `(2 . in)` specifies a width of 2 inches, while `(0.5 .` *`image`*`)` specifies half the width (or height) of the specified *image* (which should be given by its image spec).

The form `(+` *`expr`* `...)` adds up the value of the expressions. The form `(-` *`expr`* `...)` negates or subtracts the value of the expressions.

Text shown in the header line that uses `:align-to` display specifications is not automatically realigned when `display-line-numbers-mode` is turned on and off, or when the width of line numbers on display changes. To arrange for the header-line text alignment to be updated, thus keeping the header-line text aligned with the buffer text, turn on the `header-line-indent-mode` in the buffer and use its two variables, `header-line-indent` and `header-line-indent-width`, in the display specification. See Section 24.4.7 [Header Lines], page 560. Here's a simple example:

```
(setq header-line-format
      (concat (propertize " "
                          'display
                          '(space :align-to
                                  (+ header-line-indent-width 10)))
              "Column"))
```

This will keep the text '`Column`' on the header line aligned with column 10 of buffer text, regardless of whether `display-line-numbers-mode` is on or off, and also when line-number display changes its width.

41.16.4 Other Display Specifications

Here are the other sorts of display specifications that you can use in the `display` text property.

`string` Display *string* instead of the text that has this property.

Recursive display specifications are not supported—*string*'s `display` properties, if any, are not used.

`(image .` *`image-props`*`)`

This kind of display specification is an image descriptor (see Section 41.17.2 [Image Descriptors], page 1212). When used as a display specification, it means to display the image instead of the text that has the display specification.

`(slice` *x y width height*`)`
: This specification together with `image` specifies a *slice* (a partial area) of the image to display. The elements *y* and *x* specify the top left corner of the slice, within the image; *width* and *height* specify the width and height of the slice. Integers are numbers of pixels. A floating-point number in the range 0.0–1.0 stands for that fraction of the width or height of the entire image.

`((margin nil)` *string*`)`
: A display specification of this form means to display *string* instead of the text that has the display specification, at the same position as that text. It is equivalent to using just *string*, but it is done as a special case of marginal display (see Section 41.16.5 [Display Margins], page 1209).

`(left-fringe` *bitmap* [*face*]`)`
`(right-fringe` *bitmap* [*face*]`)`
: This display specification on any character of a line of text causes the specified *bitmap* be displayed in the left or right fringes for that line, instead of the characters that have the display specification. The optional *face* specifies the face whose colors are to be used for the bitmap display. See Section 41.13.4 [Fringe Bitmaps], page 1197, for the details.

`(space-width` *factor*`)`
: This display specification affects all the space characters within the text that has the specification. It displays all of these spaces *factor* times as wide as normal. The element *factor* should be an integer or float. Characters other than spaces are not affected at all; in particular, this has no effect on tab characters.

`(min-width (`*width*`))`
: This display specification ensures the text that has it takes at least *width* space on display, by adding a stretch of white space to the end of the text if the text is shorter than *width*. The text is partitioned using the identity of the parameter, which is why the parameter is a list with one element. For instance:

```
(insert (propertize "foo" 'display '(min-width (6.0))))
```

  This will add padding after '`foo`' bringing the total width up to the width of six normal characters. Note that the affected characters are identified by the `(6.0)` list in the display property, compared with `eq`. The element *width* can be either an integer or a float specifying the required minimum width of the text (see Section 41.16.3 [Pixel Specification], page 1206).

`(height` *height*`)`
: This display specification makes the text taller or shorter. Here are the possibilities for *height*:

  `(+` *n*`)`
  : This means to use a font that is *n* steps larger. A *step* is defined by the set of available fonts—specifically, those that match what was otherwise specified for this text, in all attributes except height. Each size for which a suitable font is available counts as another step. *n* should be an integer.

  `(-` *n*`)`
  : This means to use a font that is *n* steps smaller.

a number, *factor*
: A number, *factor*, means to use a font that is *factor* times as tall as the default font.

a symbol, *function*
: A symbol is a function to compute the height. It is called with the current height as argument, and should return the new height to use.

anything else, *form*
: If the *height* value doesn't fit the previous possibilities, it is a form. Emacs evaluates it to get the new height, with the symbol `height` bound to the current specified font height.

`(raise` *factor*`)`
: This kind of display specification raises or lowers the text it applies to, relative to the baseline of the line. It is mainly meant to support display of subscripts and superscripts.

  The *factor* must be a number, which is interpreted as a multiple of the height of the affected text. If it is positive, that means to display the characters raised. If it is negative, that means to display them lower down.

  Note that if the text also has a `height` display specification, which was specified before (i.e. to the left of) `raise`, the latter will affect the amount of raising or lowering in pixels, because that is based on the height of the text being raised. Therefore, if you want to display a sub- or superscript that is smaller than the normal text height, consider specifying `raise` before `height`.

You can make any display specification conditional. To do that, package it in another list of the form `(when` *condition* `.` *spec*`)`. Then the specification *spec* applies only when *condition* evaluates to a non-`nil` value. During the evaluation, `object` is bound to the string or buffer having the conditional `display` property. `position` and `buffer-position` are bound to the position within `object` and the buffer position where the `display` property was found, respectively. Both positions can be different when `object` is a string.

Note that *condition* will only be evaluated when redisplay examines the text where this display spec is located, so this feature is best suited for conditions that are relatively stable, i.e. yield, for each particular buffer position, the same results on every evaluation. If the results change for the same text location, e.g., if the result depends on the position of point, then the conditional specification might not do what you want, because redisplay examines only those parts of buffer text where it has reasons to assume that something changed since the last display cycle.

41.16.5 Displaying in the Margins

A buffer can have blank areas called *display margins* on the left and on the right. Ordinary text never appears in these areas, but you can put things into the display margins using the `display` property. There is currently no way to make text or images in the margin mouse-sensitive.

The way to display something in the margins is to specify it in a margin display specification in the `display` property of some text. This is a replacing display specification,

meaning that the text you put it on does not get displayed; the margin display appears, but that text does not.

A margin display specification looks like `((margin right-margin) spec)` or `((margin left-margin) spec)`. Here, *spec* is another display specification that says what to display in the margin. Typically it is a string of text to display, or an image descriptor.

To display something in the margin *in association with* certain buffer text, without altering or preventing the display of that text, put on that text an overlay with a `before-string` property, and put the margin display specification on the contents of the before-string.

Note that if the string to be displayed in the margin doesn't specify a face, its face is determined using the same rules and priorities as it is for strings displayed in the text area (see Section 41.12.4 [Displaying Faces], page 1179). If this results in undesirable "leaking" of faces into the margin, make sure the string has an explicit face specified for it.

Before the display margins can display anything, you must give them a nonzero width. The usual way to do that is to set these variables:

`left-margin-width` [Variable]
: This variable specifies the width of the left margin, in character cell (a.k.a. "column") units. It is buffer-local in all buffers. A value of `nil` means no left marginal area.

`right-margin-width` [Variable]
: This variable specifies the width of the right margin, in character cell units. It is buffer-local in all buffers. A value of `nil` means no right marginal area.

Setting these variables does not immediately affect the window. These variables are checked when a new buffer is displayed in the window. Thus, you can make changes take effect by calling `set-window-buffer`. Do not use these variables to try to determine the current width of the left or right margin. Instead, use the function `window-margins`.

You can also set the margin widths immediately.

`set-window-margins` *window left* **&optional** *right* [Function]
: This function specifies the margin widths for window *window*, in character cell units. The argument *left* controls the left margin, and *right* controls the right margin (default `0`).

  If *window* is not large enough to accommodate margins of the desired width, this leaves the margins of *window* unchanged.

  The values specified here may be later overridden by invoking `set-window-buffer` (see Section 29.11 [Buffers and Windows], page 722) on *window* with its *keep-margins* argument `nil` or omitted.

`window-margins` **&optional** *window* [Function]
: This function returns the width of the left and right margins of *window* as a cons cell of the form `(left . right)`. If one of the two marginal areas does not exist, its width is returned as `nil`; if neither of the two margins exist, the function returns `(nil)`. If *window* is `nil`, the selected window is used.

41.17 Images

To display an image in an Emacs buffer, you must first create an image descriptor, then use it as a display specifier in the `display` property of text that is displayed (see Section 41.16 [Display Property], page 1203).

Emacs is usually able to display images when it is run on a graphical terminal. Images cannot be displayed in a text terminal, on certain graphical terminals that lack the support for this, or if Emacs is compiled without image support. You can use the function `display-images-p` to determine if images can in principle be displayed (see Section 30.26 [Display Feature Testing], page 852).

41.17.1 Image Formats

Emacs can display a number of different image formats. Some of these image formats are supported only if particular support libraries are installed. On some platforms, Emacs can load support libraries on demand; if so, the variable `dynamic-library-alist` can be used to modify the set of known names for these dynamic libraries. See Section 42.21 [Dynamic Libraries], page 1304.

Supported image formats (and the required support libraries) include PBM and XBM (which do not depend on support libraries and are always available), XPM (`libXpm`), GIF (`libgif` or `libungif`), JPEG (`libjpeg`), TIFF (`libtiff`), PNG (`libpng`), SVG (`librsvg`), and WebP (`libwebp`).

Each of these image formats is associated with an *image type symbol*. The symbols for the above formats are, respectively, `pbm`, `xbm`, `xpm`, `gif`, `jpeg`, `tiff`, `png`, `svg`, and `webp`.

On some platforms, the built-in image support that doesn't require any optional libraries includes BMP images.[3]

Furthermore, if you build Emacs with ImageMagick (`libMagickWand`) support, Emacs can display any image format that ImageMagick can. See Section 41.17.5 [ImageMagick Images], page 1217. All images displayed via ImageMagick have type symbol `imagemagick`.

`image-types` [Variable]

This variable contains a list of type symbols for image formats which are potentially supported in the current configuration.

"Potentially" means that Emacs knows about the image types, not necessarily that they can be used (for example, they could depend on unavailable dynamic libraries). To know which image types are really available, use `image-type-available-p`.

`image-type-available-p` *type* [Function]

This function returns non-`nil` if images of type *type* can be loaded and displayed. *type* must be an image type symbol.

For image types whose support libraries are statically linked, this function always returns `t`. For image types whose support libraries are dynamically loaded, it returns `t` if the library could be loaded and `nil` otherwise.

[3] On MS-Windows, this requires `w32-use-native-image-API` to be set non-`nil`.

41.17.2 Image Descriptors

An *image descriptor* is a list which specifies the underlying data for an image, and how to display it. It is typically used as the value of a `display` overlay or text property (see Section 41.16.4 [Other Display Specs], page 1207); but See Section 41.17.9 [Showing Images], page 1227, for convenient helper functions to insert images into buffers.

Each image descriptor has the form `(image . props)`, where *props* is a property list of alternating keyword symbols and values, including at least the pair `:type type` that specifies the image type.

Image descriptors which define image dimensions, `:width`, `:height`, `:max-width` and `:max-height`, may take either an integer, which represents the dimension in pixels, or a pair `(value . em)`, where *value* is the dimension's length in *ems*[4]. One em is equivalent to the height of the font and *value* may be an integer or a float.

The following is a list of properties that are meaningful for all image types (there are also properties which are meaningful only for certain image types, as documented in the following subsections):

`:type type`
: The image type. Every image descriptor must include this property.

`:file file`
: This says to load the image from file *file*. If *file* is not an absolute file name, it is expanded relative to each of the directories mentioned by `image-load-path` (see Section 41.17.8 [Defining Images], page 1225).

`:data data`
: This specifies the raw image data. Each image descriptor must have either `:data` or `:file`, but not both.

  For most image types, the value of a `:data` property should be a string containing the image data. Some image types do not support `:data`; for some others, `:data` alone is not enough, so you need to use other image properties along with `:data`. See the following subsections for details.

`:margin margin`
: This specifies how many pixels to add as an extra margin around the image. The value, *margin*, must be a non-negative number, or a pair `(x . y)` of such numbers. If it is a pair, *x* specifies how many pixels to add horizontally, and *y* specifies how many pixels to add vertically. If `:margin` is not specified, the default is zero.

`:ascent ascent`
: This specifies the amount of the image's height to use for its ascent—that is, the part above the baseline. The value, *ascent*, must be a number in the range 0 to 100, or the symbol `center`.

  If *ascent* is a number, that percentage of the image's height is used for its ascent.

  If *ascent* is `center`, the image is vertically centered around a centerline which would be the vertical centerline of text drawn at the position of the image,

[4] In typography an em is a distance equivalent to the height of the type. For example when using 12 point type 1 em is equal to 12 points. Its use ensures distances and type remain proportional.

in the manner specified by the text properties and overlays that apply to the image.

If this property is omitted, it defaults to 50.

`:relief` *relief*
: This adds a shadow rectangle around the image. The value, *relief*, specifies the width of the shadow lines, in pixels. If *relief* is negative, shadows are drawn so that the image appears as a pressed button; otherwise, it appears as an unpressed button.

`:width` *width*, `:height` *height*
: The `:width` and `:height` keywords are used for scaling the image. If only one of them is specified, the other one will be calculated so as to preserve the aspect ratio. If both are specified, aspect ratio may not be preserved.

`:max-width` *max-width*, `:max-height` *max-height*
: The `:max-width` and `:max-height` keywords are used for scaling if the size of the image exceeds these values. If `:width` is set, it will have precedence over `max-width`, and if `:height` is set, it will have precedence over `max-height`, but you can otherwise mix these keywords as you wish.

  If both `:max-width` and `:height` are specified, but `:width` is not, preserving the aspect ratio might require that width exceeds `:max-width`. If this happens, scaling will use a smaller value for the height so as to preserve the aspect ratio while not exceeding `:max-width`. Similarly when both `:max-height` and `:width` are specified, but `:height` is not. For example, if you have a 200x100 image and specify that `:width` should be 400 and `:max-height` should be 150, you'll end up with an image that is 300x150: Preserving the aspect ratio and not exceeding the "max" setting. This combination of parameters is a useful way of saying "display this image as large as possible, but no larger than the available display area".

`:scale` *scale*
: This should be a number, where values higher than 1 means to increase the size, and lower means to decrease the size, by multiplying both the width and height. For instance, a value of 0.25 will make the image a quarter size of what it originally was. If the scaling makes the image larger than specified by `:max-width` or `:max-height`, the resulting size will not exceed those two values. If both `:scale` and `:height`/`:width` are specified, the height/width will be adjusted by the specified scaling factor.

`:rotation` *angle*
: Specifies a rotation angle in degrees. Only multiples of 90 degrees are supported, unless the image type is `imagemagick`. Positive values rotate clockwise, negative values counter-clockwise. Rotation is performed after scaling and cropping.

`:flip` *flip*
: If this is `t`, the image will be horizontally flipped. Currently it has no effect if the image type is `imagemagick`. Vertical flipping can be achieved by rotating the image 180 degrees and toggling this value.

`:transform-smoothing` *smooth*

If this is `t`, any image transform will have smoothing applied; if `nil`, no smoothing will be applied. The exact algorithm used is platform dependent, but should be equivalent to bilinear filtering. Disabling smoothing will use the nearest neighbor algorithm.

If this property is not specified, `create-image` will use the `image-transform-smoothing` user option to say whether scaling should be done or not. This option can be `nil` (no smoothing), `t` (use smoothing) or a predicate function that's called with the image object as the only parameter, and should return either `nil` or `t`. The default is for down-scaling to apply smoothing, and for large up-scaling to not apply smoothing.

`:index` *frame*

See Section 41.17.10 [Multi-Frame Images], page 1229.

`:conversion` *algorithm*

This specifies a conversion algorithm that should be applied to the image before it is displayed; the value, *algorithm*, specifies which algorithm.

`laplace`

`emboss` Specifies the Laplace edge detection algorithm, which blurs out small differences in color while highlighting larger differences. People sometimes consider this useful for displaying the image for a disabled button.

`(edge-detection :matrix` *matrix* `:color-adjust` *adjust*`)`

Specifies a general edge-detection algorithm. *matrix* must be either a nine-element list or a nine-element vector of numbers. A pixel at position x/y in the transformed image is computed from original pixels around that position. *matrix* specifies, for each pixel in the neighborhood of x/y, a factor with which that pixel will influence the transformed pixel; element 0 specifies the factor for the pixel at $x-1/y-1$, element 1 the factor for the pixel at $x/y-1$ etc., as shown below:

$$\begin{pmatrix} x-1/y-1 & x/y-1 & x+1/y-1 \\ x-1/y & x/y & x+1/y \\ x-1/y+1 & x/y+1 & x+1/y+1 \end{pmatrix}$$

The resulting pixel is computed from the color intensity of the color resulting from summing up the RGB values of surrounding pixels, multiplied by the specified factors, and dividing that sum by the sum of the factors' absolute values.

Laplace edge-detection currently uses a matrix of

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -1 \end{pmatrix}$$

Emboss edge-detection uses a matrix of

$$\begin{pmatrix} 2 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & -2 \end{pmatrix}$$

`disabled` Specifies transforming the image so that it looks disabled.

`:mask` *`mask`*

If *mask* is `heuristic` or `(heuristic` *`bg`*`)`, build a clipping mask for the image, so that the background of a frame is visible behind the image. If *bg* is not specified, or if *bg* is `t`, determine the background color of the image by looking at the four corners of the image, assuming the most frequently occurring color from the corners is the background color of the image. Otherwise, *bg* must be a list `(`*`red green blue`*`)` specifying the color to assume for the background of the image.

If *mask* is `nil`, remove a mask from the image, if it has one. Images in some formats include a mask which can be removed by specifying `:mask nil`.

`:pointer` *`shape`*

This specifies the pointer shape when the mouse pointer is over this image. See Section 30.19 [Pointer Shape], page 841, for available pointer shapes.

`:map` *`map`* This associates an image map of *hot spots* with this image.

An image map is an alist where each element has the format `(`*`area id plist`*`)`. An *area* is specified as either a rectangle, a circle, or a polygon.

A rectangle is a cons `(rect . ((`*`x0 . y0`*`) . (`*`x1 . y1`*`)))` which specifies the pixel coordinates of the upper left and bottom right corners of the rectangle area.

A circle is a cons `(circle . ((`*`x0 . y0`*`) . `*`r`*`))` which specifies the center and the radius of the circle; *r* may be a float or integer.

A polygon is a cons `(poly . [`*`x0 y0 x1 y1 ...`*`])` where each pair in the vector describes one corner in the polygon.

When the mouse pointer lies on a hot-spot area of an image, the *plist* of that hot-spot is consulted; if it contains a `help-echo` property, that defines a tool-tip for the hot-spot, and if it contains a `pointer` property, that defines the shape of the mouse cursor when it is on the hot-spot. See Section 30.19 [Pointer Shape], page 841, for available pointer shapes.

When you click the mouse when the mouse pointer is over a hot-spot, an event is composed by combining the *id* of the hot-spot with the mouse event; for instance, `[area4 mouse-1]` if the hot-spot's *id* is `area4`.

Note that the map's coordinates should reflect the displayed image after all transforms have been done (rotation, scaling and so on), and also note that Emacs (by default) performs auto-scaling of images, so to make things match up, you should either specify `:scale 1.0` when creating the image, or use the result of `image-compute-scaling-factor` to compute the elements of the map.

`image-mask-p` *spec* **&optional** *frame* [Function]
: This function returns `t` if image *spec* has a mask bitmap. *frame* is the frame on which the image will be displayed. *frame* `nil` or omitted means to use the selected frame (see Section 30.10 [Input Focus], page 827).

`image-transforms-p` **&optional** *frame* [Function]
: This function returns non-`nil` if *frame* supports image scaling and rotation. *frame* `nil` or omitted means to use the selected frame (see Section 30.10 [Input Focus], page 827). The returned list includes symbols that indicate which image transform operations are supported:

  `scale` Image scaling is supported by *frame* via the `:scale`, `:width`, `:height`, `:max-width`, and `:max-height` properties.

  `rotate90` Image rotation is supported by *frame* if the rotation angle is an integral multiple of 90 degrees.

  If image transforms are not supported, `:rotation`, `:crop`, `:width`, `:height`, `:scale`, `:max-width` and `:max-height` will only be usable through ImageMagick, if available (see Section 41.17.5 [ImageMagick Images], page 1217).

41.17.3 XBM Images

To use XBM format, specify `xbm` as the image type. This image format doesn't require an external library, so images of this type are always supported.

Additional image properties supported for the `xbm` image type are:

`:foreground` *foreground*
: The value, *foreground*, should be a string specifying the image foreground color, or `nil` for the default color. This color is used for each pixel in the XBM that is 1. The default is the frame's foreground color.

`:background` *background*
: The value, *background*, should be a string specifying the image background color, or `nil` for the default color. This color is used for each pixel in the XBM that is 0. The default is the frame's background color.

If you specify an XBM image using data within Emacs instead of an external file, use the following three properties:

`:data` *data*
: The value, *data*, specifies the contents of the image. There are three formats you can use for *data*:

  - A vector of strings or bool-vectors, each specifying one line of the image. Do specify `:data-height` and `:data-width`.
  - A string containing the same byte sequence as an XBM file would contain.
  - A string or a bool-vector containing the bits of the image (plus perhaps some extra bits at the end that will not be used). It should contain at least *stride* `*` *height* bits, where *stride* is the smallest multiple of 8 greater than or equal to the width of the image. In this case, you should specify `:data-height`, `:data-width` and `:stride`, both to indicate that the string

contains just the bits rather than a whole XBM file, and to specify the size of the image.

`:stride` *stride*
: The number of bool vector entries stored for each row; the smallest multiple of 8 greater than or equal to *width*.

41.17.4 XPM Images

To use XPM format, specify `xpm` as the image type. The additional image property `:color-symbols` is also meaningful with the `xpm` image type:

`:color-symbols` *symbols*
: The value, *symbols*, should be an alist whose elements have the form (*name* . *color*). In each element, *name* is the name of a color as it appears in the image file, and *color* specifies the actual color to use for displaying that name.

41.17.5 ImageMagick Images

If your Emacs build has ImageMagick support, you can use the ImageMagick library to load many image formats (see Section "File Conveniences" in *The GNU Emacs Manual*). The image type symbol for images loaded via ImageMagick is `imagemagick`, regardless of the actual underlying image format.

To check for ImageMagick support, use the following:

```
(image-type-available-p 'imagemagick)
```

`imagemagick-types` [Function]
: This function returns a list of image file extensions supported by the current ImageMagick installation. Each list element is a symbol representing an internal ImageMagick name for an image type, such as `BMP` for `.bmp` images.

`imagemagick-enabled-types` [User Option]
: The value of this variable is a list of ImageMagick image types which Emacs may attempt to render using ImageMagick. Each list element should be one of the symbols in the list returned by `imagemagick-types`, or an equivalent string. Alternatively, a value of `t` enables ImageMagick for all possible image types. Regardless of the value of this variable, `imagemagick-types-inhibit` (see below) takes precedence.

`imagemagick-types-inhibit` [User Option]
: The value of this variable lists the ImageMagick image types which should never be rendered using ImageMagick, regardless of the value of `imagemagick-enabled-types`. A value of `t` disables ImageMagick entirely.

`image-format-suffixes` [Variable]
: This variable is an alist mapping image types to file name extensions. Emacs uses this in conjunction with the `:format` image property (see below) to give a hint to the ImageMagick library as to the type of an image. Each element has the form (*type extension*), where *type* is a symbol specifying an image content-type, and *extension* is a string that specifies the associated file name extension.

Images loaded with ImageMagick support the following additional image descriptor properties:

`:background` *background*
: *background*, if non-`nil`, should be a string specifying a color, which is used as the image's background color if the image supports transparency. If the value is `nil`, it defaults to the frame's background color.

`:format` *type*
: The value, *type*, should be a symbol specifying the type of the image data, as found in `image-format-suffixes`. This is used when the image does not have an associated file name, to provide a hint to ImageMagick to help it detect the image type.

`:crop` *geometry*
: The value of *geometry* should be a list of the form `(`*width height x y*`)`. *width* and *height* specify the width and height of the cropped image. If *x* is a positive number it specifies the offset of the cropped area from the left of the original image, and if negative the offset from the right. If *y* is a positive number it specifies the offset from the top of the original image, and if negative from the bottom. If *x* or *y* are `nil` or unspecified the crop area will be centered on the original image.

  If the crop area is outside or overlaps the edge of the image it will be reduced to exclude any areas outside of the image. This means it is not possible to use `:crop` to increase the size of the image by entering large *width* or *height* values.

  Cropping is performed after scaling but before rotation.

41.17.6 SVG Images

SVG (Scalable Vector Graphics) is an XML format for specifying images. SVG images support the following additional image descriptor properties:

`:foreground` *foreground*
: *foreground*, if non-`nil`, should be a string specifying a color, which is used as the image's foreground color. If the value is `nil`, it defaults to the current face's foreground color.

`:background` *background*
: *background*, if non-`nil`, should be a string specifying a color, which is used as the image's background color if the image supports transparency. If the value is `nil`, it defaults to the current face's background color.

`:css` *css*
: *css*, if non-`nil`, should be a string specifying the CSS to override the default CSS used when generating the image.

SVG library

If your Emacs build has SVG support, you can create and manipulate these images with the following functions from the `svg.el` library.

`svg-create` *width height* **&rest** *args* [Function]
: Create a new, empty SVG image with the specified dimensions. *args* is an argument plist with you can specify following:

`:stroke-width`
: The default width (in pixels) of any lines created.

`:stroke` The default stroke color on any lines created.

This function returns an *SVG object*, a Lisp data structure that specifies an SVG image, and all the following functions work on that structure. The argument *svg* in the following functions specifies such an SVG object.

`svg-gradient` *svg id type stops* [Function]

Create a gradient in *svg* with identifier *id*. *type* specifies the gradient type, and can be either `linear` or `radial`. *stops* is a list of percentage/color pairs.

The following will create a linear gradient that goes from red at the start, to green 25% of the way, to blue at the end:

```
(svg-gradient svg "gradient1" 'linear
              '((0 . "red") (25 . "green") (100 . "blue")))
```

The gradient created (and inserted into the SVG object) can later be used by all functions that create shapes.

All the following functions take an optional list of keyword parameters that alter the various attributes from their default values. Valid attributes include:

`:stroke-width`
: The width (in pixels) of lines drawn, and outlines around solid shapes.

`:stroke-color`
: The color of lines drawn, and outlines around solid shapes.

`:fill-color`
: The color used for solid shapes.

`:id` The identified of the shape.

`:gradient`
: If given, this should be the identifier of a previously defined gradient object.

`:clip-path`
: Identifier of a clip path.

`svg-rectangle` *svg x y width height* **&rest** *args* [Function]

Add to *svg* a rectangle whose upper left corner is at position *x*/*y* and whose size is *width*/*height*.

```
(svg-rectangle svg 100 100 500 500 :gradient "gradient1")
```

`svg-circle` *svg x y radius* **&rest** *args* [Function]

Add to *svg* a circle whose center is at *x*/*y* and whose radius is *radius*.

`svg-ellipse` *svg x y x-radius y-radius* **&rest** *args* [Function]

Add to *svg* an ellipse whose center is at *x*/*y*, and whose horizontal radius is *x-radius* and the vertical radius is *y-radius*.

`svg-line` *svg x1 y1 x2 y2* **&rest** *args* [Function]

Add to *svg* a line that starts at *x1*/*y1* and extends to *x2*/*y2*.

`svg-polyline` *svg points* **&rest** *args* [Function]

Add to *svg* a multiple-segment line (a.k.a. "polyline") that goes through *points*, which is a list of X/Y position pairs.

```
(svg-polyline svg '((200 . 100) (500 . 450) (80 . 100))
              :stroke-color "green")
```

`svg-polygon` *svg points* **&rest** *args* [Function]

Add a polygon to *svg* where *points* is a list of X/Y pairs that describe the outer circumference of the polygon.

```
(svg-polygon svg '((100 . 100) (200 . 150) (150 . 90))
             :stroke-color "blue" :fill-color "red")
```

`svg-path` *svg commands* **&rest** *args* [Function]

Add the outline of a shape to *svg* according to *commands*, see [SVG Path Commands], page 1222.

Coordinates by default are absolute. To use coordinates relative to the last position, or – initially – to the origin, set the attribute *:relative* to `t`. This attribute can be specified for the function or for individual commands. If specified for the function, then all commands use relative coordinates by default. To make an individual command use absolute coordinates, set *:relative* to `nil`.

```
(svg-path svg
  '((moveto ((100 . 100)))
    (lineto ((200 . 0) (0 . 200) (-200 . 0)))
    (lineto ((100 . 100)) :relative nil))
  :stroke-color "blue"
  :fill-color "lightblue"
  :relative t)
```

`svg-text` *svg text* **&rest** *args* [Function]

Add the specified *text* to *svg*.

```
(svg-text
 svg "This is a text"
 :font-size "40"
 :font-weight "bold"
 :stroke "black"
 :fill "white"
 :font-family "impact"
 :letter-spacing "4pt"
 :x 300
 :y 400
 :stroke-width 1)
```

`svg-embed` *svg image image-type datap* **&rest** *args* [Function]

Add an embedded (raster) image to *svg*. If *datap* is `nil`, *image* should be a file name; otherwise it should be a string containing the image data as raw bytes. *image-type* should be a MIME image type, for instance `"image/jpeg"`.

```
(svg-embed svg "~/rms.jpg" "image/jpeg" nil
```

```
                :width "100px" :height "100px"
                :x "50px" :y "75px")
```

`svg-embed-base-uri-image` *svg relative-filename* **&rest** *args* [Function]

To *svg* add an embedded (raster) image placed at *relative-filename*. *relative-filename* is searched inside `file-name-directory` of the `:base-uri` svg image property. `:base-uri` specifies a (possibly non-existing) file name of the svg image to be created, thus all the embedded files are searched relatively to the `:base-uri` filename's directory. If `:base-uri` is omitted, then filename from where svg image is loaded is used. Using `:base-uri` improves the performance of embedding large images, comparing to `svg-embed`, because all the work is done directly by librsvg.

```
;; Embedding /tmp/subdir/rms.jpg and /tmp/another/rms.jpg
(svg-embed-base-uri-image svg "subdir/rms.jpg"
            :width "100px" :height "100px"
            :x "50px" :y "75px")
(svg-embed-base-uri-image svg "another/rms.jpg"
            :width "100px" :height "100px"
            :x "75px" :y "50px")
(svg-image svg :scale 1.0
            :base-uri "/tmp/dummy"
            :width 175 :height 175)
```

`svg-clip-path` *svg* **&rest** *args* [Function]

Add a clipping path to *svg*. If applied to a shape via the *:clip-path* property, parts of that shape which lie outside of the clipping path are not drawn.

```
(let ((clip-path (svg-clip-path svg :id "foo")))
  (svg-circle clip-path 200 200 175))
(svg-rectangle svg 50 50 300 300
               :fill-color "red"
               :clip-path "url(#foo)")
```

`svg-node` *svg tag* **&rest** *args* [Function]

Add the custom node *tag* to *svg*.

```
(svg-node svg
          'rect
          :width 300 :height 200 :x 50 :y 100 :fill-color "green")
```

`svg-remove` *svg id* [Function]

Remove the element with identifier `id` from the `svg`.

`svg-image` *svg* [Function]

Finally, the `svg-image` takes an SVG object as its argument and returns an image object suitable for use in functions like `insert-image`.

Here's a complete example that creates and inserts an image with a circle:

```
(let ((svg (svg-create 400 400 :stroke-width 10)))
  (svg-gradient svg "gradient1" 'linear '((0 . "red") (100 . "blue")))
  (svg-circle svg 200 200 100 :gradient "gradient1"
```

```
                    :stroke-color "green")
    (insert-image (svg-image svg)))
```

SVG Path Commands

SVG paths allow creation of complex images by combining lines, curves, arcs, and other basic shapes. The functions described below allow invoking SVG path commands from a Lisp program.

`moveto` *points* [Command]

Move the pen to the first point in *points*. Additional points are connected with lines. *points* is a list of X/Y coordinate pairs. Subsequent `moveto` commands represent the start of a new *subpath*.

```
(svg-path svg '((moveto ((200 . 100) (100 . 200) (0 . 100))))
          :fill "white" :stroke "black")
```

`closepath` [Command]

End the current subpath by connecting it back to its initial point. A line is drawn along the connection.

```
(svg-path svg '((moveto ((200 . 100) (100 . 200) (0 . 100)))
                (closepath)
                (moveto ((75 . 125) (100 . 150) (125 . 125)))
                (closepath))
          :fill "red" :stroke "black")
```

`lineto` *points* [Command]

Draw a line from the current point to the first element in *points*, a list of X/Y position pairs. If more than one point is specified, draw a polyline.

```
(svg-path svg '((moveto ((200 . 100)))
                (lineto ((100 . 200) (0 . 100))))
          :fill "yellow" :stroke "red")
```

`horizontal-lineto` *x-coordinates* [Command]

Draw a horizontal line from the current point to the first element in *x-coordinates*. Specifying multiple coordinates is possible, although this usually doesn't make sense.

```
(svg-path svg '((moveto ((100 . 200)))
                (horizontal-lineto (300)))
          :stroke "green")
```

`vertical-lineto` *y-coordinates* [Command]

Draw vertical lines.

```
(svg-path svg '((moveto ((200 . 100)))
                (vertical-lineto (300)))
          :stroke "green")
```

`curveto` *coordinate-sets* [Command]

Using the first element in *coordinate-sets*, draw a cubic Bézier curve from the current point. If there are multiple coordinate sets, draw a polybezier. Each coordinate set is

a list of the form `(x1 y1 x2 y2 x y)`, where (*x*, *y*) is the curve's end point. (*x1*, *y1*) and (*x2*, *y2*) are control points at the beginning and at the end, respectively.

```
(svg-path svg '((moveto ((100 . 100)))
                (curveto ((200 100 100 200 200 200)
                          (300 200 0 100 100 100))))
          :fill "transparent" :stroke "red")
```

`smooth-curveto` *coordinate-sets* [Command]

Using the first element in *coordinate-sets*, draw a cubic Bézier curve from the current point. If there are multiple coordinate sets, draw a polybezier. Each coordinate set is a list of the form `(x2 y2 x y)`, where (*x*, *y*) is the curve's end point and (*x2*, *y2*) is the corresponding control point. The first control point is the reflection of the second control point of the previous command relative to the current point, if that command was `curveto` or `smooth-curveto`. Otherwise the first control point coincides with the current point.

```
(svg-path svg '((moveto ((100 . 100)))
                (curveto ((200 100 100 200 200 200)))
                (smooth-curveto ((0 100 100 100))))
          :fill "transparent" :stroke "blue")
```

`quadratic-bezier-curveto` *coordinate-sets* [Command]

Using the first element in *coordinate-sets*, draw a quadratic Bézier curve from the current point. If there are multiple coordinate sets, draw a polybezier. Each coordinate set is a list of the form `(x1 y1 x y)`, where (*x*, *y*) is the curve's end point and (*x1*, *y1*) is the control point.

```
(svg-path svg '((moveto ((200 . 100)))
                (quadratic-bezier-curveto ((300 100 300 200)))
                (quadratic-bezier-curveto ((300 300 200 300)))
                (quadratic-bezier-curveto ((100 300 100 200)))
                (quadratic-bezier-curveto ((100 100 200 100))))
          :fill "transparent" :stroke "pink")
```

`smooth-quadratic-bezier-curveto` *coordinate-sets* [Command]

Using the first element in *coordinate-sets*, draw a quadratic Bézier curve from the current point. If there are multiple coordinate sets, draw a polybezier. Each coordinate set is a list of the form `(x y)`, where (*x*, *y*) is the curve's end point. The control point is the reflection of the control point of the previous command relative to the current point, if that command was `quadratic-bezier-curveto` or `smooth-quadratic-bezier-curveto`. Otherwise the control point coincides with the current point.

```
(svg-path svg '((moveto ((200 . 100)))
                (quadratic-bezier-curveto ((300 100 300 200)))
                (smooth-quadratic-bezier-curveto ((200 300)))
                (smooth-quadratic-bezier-curveto ((100 200)))
                (smooth-quadratic-bezier-curveto ((200 100))))
          :fill "transparent" :stroke "lightblue")
```

`elliptical-arc` *coordinate-sets* [Command]

Using the first element in *coordinate-sets*, draw an elliptical arc from the current point. If there are multiple coordinate sets, draw a sequence of elliptical arcs. Each coordinate set is a list of the form `(rx ry x y)`, where (*x*, *y*) is the end point of the ellipse, and (*rx*, *ry*) are its radii. Attributes may be appended to the list:

`:x-axis-rotation`
: The angle in degrees by which the x-axis of the ellipse is rotated relative to the x-axis of the current coordinate system.

`:large-arc`
: If set to `t`, draw an arc sweep greater than or equal to 180 degrees. Otherwise, draw an arc sweep smaller than or equal to 180 degrees.

`:sweep` If set to `t`, draw an arc in *positive angle direction*. Otherwise, draw it in *negative angle direction*.

```
(svg-path svg '((moveto ((200 . 250)))
                (elliptical-arc ((75 75 200 350))))
          :fill "transparent" :stroke "red")
(svg-path svg '((moveto ((200 . 250)))
                (elliptical-arc ((75 75 200 350 :large-arc t))))
          :fill "transparent" :stroke "green")
(svg-path svg '((moveto ((200 . 250)))
                (elliptical-arc ((75 75 200 350 :sweep t))))
          :fill "transparent" :stroke "blue")
(svg-path svg '((moveto ((200 . 250)))
                (elliptical-arc ((75 75 200 350 :large-arc t
                                        :sweep t))))
          :fill "transparent" :stroke "gray")
(svg-path svg '((moveto ((160 . 100)))
                (elliptical-arc ((40 100 80 0)))
                (elliptical-arc ((40 100 -40 -70
                                        :x-axis-rotation -120)))
                (elliptical-arc ((40 100 -40 70
                                        :x-axis-rotation -240))))
          :stroke "pink" :fill "lightblue"
          :relative t)
```

41.17.7 Other Image Types

For PBM images, specify image type `pbm`. Color, gray-scale and monochromatic images are supported. For mono PBM images, two additional image properties are supported.

`:foreground` *foreground*
: The value, *foreground*, should be a string specifying the image foreground color, or `nil` for the default color. This color is used for each pixel in the PBM that is 1. The default is the frame's foreground color.

`:background` *background*
: The value, *background*, should be a string specifying the image background color, or `nil` for the default color. This color is used for each pixel in the PBM that is 0. The default is the frame's background color.

The remaining image types that Emacs can support are:

GIF
: Image type `gif`. Supports the `:index` property. See Section 41.17.10 [Multi-Frame Images], page 1229.

JPEG
: Image type `jpeg`.

PNG
: Image type `png`.

TIFF
: Image type `tiff`. Supports the `:index` property. See Section 41.17.10 [Multi-Frame Images], page 1229.

WebP
: Image type `webp`.

41.17.8 Defining Images

The functions `create-image`, `defimage` and `find-image` provide convenient ways to create image descriptors.

`create-image` *file-or-data* **&optional** *type data-p* **&rest** *props* [Function]
: This function creates and returns an image descriptor which uses the data in *file-or-data*. *file-or-data* can be a file name or a string containing the image data; *data-p* should be `nil` for the former case, non-`nil` for the latter case. If *file-or-data* is a relative file name, the function will search for it in directories mentioned in `image-load-path`.

  The optional argument *type* is a symbol specifying the image type. If *type* is omitted or `nil`, `create-image` tries to determine the image type from the file's first few bytes, or else from the file's name.

  The remaining arguments, *props*, specify additional image properties—for example,

  ```
  (create-image "foo.xpm" 'xpm nil :mask 'heuristic)
  ```

  See Section 41.17.2 [Image Descriptors], page 1212, for the list of supported properties. Some properties are specific to certain image types, and are described in subsections specific to those types.

  The function returns `nil` if images of this type are not supported. Otherwise it returns an image descriptor.

`defimage` *symbol specs* **&optional** *doc* [Macro]
: This macro defines *symbol* as an image name. The arguments *specs* is a list which specifies how to display the image. The third argument, *doc*, is an optional documentation string.

  Each argument in *specs* has the form of a property list, and each one should specify at least the `:type` property and either the `:file` or the `:data` property. The value of `:type` should be a symbol specifying the image type, the value of `:file` is the file to load the image from, and the value of `:data` is a string containing the actual image data. Here is an example:

  ```
  (defimage test-image
  ```

```
((:type xpm :file "~/test1.xpm")
 (:type xbm :file "~/test1.xbm")))
```

`defimage` tests each argument, one by one, to see if it is usable—that is, if the type is supported and the file exists. The first usable argument is used to make an image descriptor which is stored in *symbol*.

If none of the alternatives will work, then *symbol* is defined as `nil`.

`image-property` *image property* [Function]

Return the value of *property* in *image*. Properties can be set by using `setf`. Setting a property to `nil` will remove the property from the image.

`find-image` *specs* [Function]

This function provides a convenient way to find an image satisfying one of a list of image specifications *specs*.

Each specification in *specs* is a property list with contents depending on image type. All specifications must at least contain the properties `:type` *type* and either `:file` *file* or `:data` *data*, where *type* is a symbol specifying the image type, e.g., `xbm`, *file* is the file to load the image from, and *data* is a string containing the actual image data. The first specification in the list whose *type* is supported, and *file* exists, is used to construct the image specification to be returned. If no specification is satisfied, `nil` is returned.

The image is looked for in `image-load-path`.

`image-load-path` [User Option]

This variable's value is a list of locations in which to search for image files. If an element is a string or a variable symbol whose value is a string, the string is taken to be the name of a directory to search. If an element is a variable symbol whose value is a list, that is taken to be a list of directories to search.

The default is to search in the `images` subdirectory of the directory specified by `data-directory`, then the directory specified by `data-directory`, and finally in the directories in `load-path`. Subdirectories are not automatically included in the search, so if you put an image file in a subdirectory, you have to supply the subdirectory explicitly. For example, to find the image `images/foo/bar.xpm` within `data-directory`, you should specify the image as follows:

```
(defimage foo-image '((:type xpm :file "foo/bar.xpm")))
```

`image-load-path-for-library` *library image* **&optional** *path no-error* [Function]

This function returns a suitable search path for images used by the Lisp package *library*.

The function searches for *image* first using `image-load-path`, excluding `data-directory/images`, and then in `load-path`, followed by a path suitable for *library*, which includes `../../etc/images` and `../etc/images` relative to the library file itself, and finally in `data-directory/images`.

Then this function returns a list of directories which contains first the directory in which *image* was found, followed by the value of `load-path`. If *path* is given, it is used instead of `load-path`.

If *no-error* is non-`nil` and a suitable path can't be found, don't signal an error. Instead, return a list of directories as before, except that `nil` appears in place of the image directory.

Here is an example of using `image-load-path-for-library`:

```
(defvar image-load-path) ; shush compiler
(let* ((load-path (image-load-path-for-library
                   "mh-e" "mh-logo.xpm"))
       (image-load-path (cons (car load-path)
                              image-load-path)))
  (mh-tool-bar-folder-buttons-init))
```

Images are automatically scaled when created based on the `image-scaling-factor` variable. The value is either a floating point number (where numbers higher than 1 means to increase the size and lower means to shrink the size), or the symbol `auto`, which will compute a scaling factor based on the font pixel size.

41.17.9 Showing Images

You can use an image descriptor by setting up the `display` property yourself, but it is easier to use the functions in this section.

`insert-image` *image* **&optional** *string area slice inhibit-isearch* [Function]

This function inserts *image* in the current buffer at point. The value *image* should be an image descriptor; it could be a value returned by `create-image`, or the value of a symbol defined with `defimage`. The argument *string* specifies the text to put in the buffer to hold the image. If it is omitted or `nil`, `insert-image` uses `" "` by default.

The argument *area* specifies whether to put the image in a margin. If it is `left-margin`, the image appears in the left margin; `right-margin` specifies the right margin. If *area* is `nil` or omitted, the image is displayed at point within the buffer's text.

The argument *slice* specifies a slice of the image to insert. If *slice* is `nil` or omitted the whole image is inserted. (However, note that images are chopped on display at the window's right edge, because wrapping images is not supported.) Otherwise, *slice* is a list `(`*x y width height*`)` which specifies the *x* and *y* positions and *width* and *height* of the image area to insert. Integer values are in units of pixels. A floating-point number in the range 0.0–1.0 stands for that fraction of the width or height of the entire image.

Internally, this function inserts *string* in the buffer, and gives it a `display` property which specifies *image*. See Section 41.16 [Display Property], page 1203. By default, doing interactive searches in the buffer will consider *string* when searching. If *inhibit-isearch* is non-`nil`, this is inhibited.

`insert-sliced-image` *image* **&optional** *string area rows cols* [Function]

This function inserts *image* in the current buffer at point, like `insert-image`, but splits the image into *rows*x*cols* equally sized slices.

Emacs displays each slice as a separate image, and allows more intuitive scrolling up/down, instead of jumping up/down the entire image when paging through a buffer that displays (large) images.

`put-image` *image pos* **&optional** *string area* [Function]
This function puts image *image* in front of *pos* in the current buffer. The argument *pos* should be an integer or a marker. It specifies the buffer position where the image should appear. The argument *string* specifies the text that should hold the image as an alternative to the default '`x`'.

The argument *image* must be an image descriptor, perhaps returned by `create-image` or stored by `defimage`.

The argument *area* specifies whether to put the image in a margin. If it is `left-margin`, the image appears in the left margin; `right-margin` specifies the right margin. If *area* is `nil` or omitted, the image is displayed at point within the buffer's text.

Internally, this function creates an overlay, and gives it a `before-string` property containing text that has a `display` property whose value is the image. (Whew! that was a mouthful...)

`remove-images` *start end* **&optional** *buffer* [Function]
This function removes images in *buffer* between positions *start* and *end*. If *buffer* is omitted or `nil`, images are removed from the current buffer.

This removes only images that were put into *buffer* the way `put-image` does it, not images that were inserted with `insert-image` or in other ways.

`image-size` *spec* **&optional** *pixels frame* [Function]
This function returns the size of an image as a pair `(`*`width`*` . `*`height`*`)`. *spec* is an image specification. *pixels* non-`nil` means return sizes measured in pixels, otherwise return sizes measured in the default character size of *frame* (see Section 30.3.2 [Frame Font], page 799). *frame* is the frame on which the image will be displayed. *frame* `nil` or omitted means use the selected frame (see Section 30.10 [Input Focus], page 827).

`max-image-size` [Variable]
This variable is used to define the maximum size of image that Emacs will load. Emacs will refuse to load (and display) any image that is larger than this limit.

If the value is an integer, it directly specifies the maximum image height and width, measured in pixels. If it is floating point, it specifies the maximum image height and width as a ratio to the frame height and width. If the value is non-numeric, there is no explicit limit on the size of images.

The purpose of this variable is to prevent unreasonably large images from accidentally being loaded into Emacs. It only takes effect the first time an image is loaded. Once an image is placed in the image cache, it can always be displayed, even if the value of `max-image-size` is subsequently changed (see Section 41.17.11 [Image Cache], page 1230).

`image-at-point-p` [Function]
This function returns `t` if point is on an image, and `nil` otherwise.

Images inserted with the insertion functions above also get a local keymap installed in the text properties (or overlays) that span the displayed image. This keymap defines the following commands:

`i +` Increase the image size (`image-increase-size`)

`i -` Decrease the image size (`image-decrease-size`).

`i r` Rotate the image (`image-rotate`).

`i h` Flip the image horizontally (`image-flip-horizontally`).

`i v` Flip the image vertically (`image-flip-vertically`).

`i o` Save the image to a file (`image-save`).

`i c` Interactively crop the image (`image-crop`).

`i x` Interactively cut a rectangle from the image (`image-cut`).

See Section "Image Mode" in *The GNU Emacs Manual*, for more details about these image-specific key bindings.

41.17.10 Multi-Frame Images

Some image files can contain more than one image. We say that there are multiple "frames" in the image. At present, Emacs supports multiple frames for GIF, TIFF, and certain ImageMagick formats such as DJVM.

The frames can be used either to represent multiple pages (this is usually the case with multi-frame TIFF files, for example), or to create animation (usually the case with multi-frame GIF files).

A multi-frame image has a property `:index`, whose value is an integer (counting from 0) that specifies which frame is being displayed.

`image-multi-frame-p` *image* [Function]
This function returns non-`nil` if *image* contains more than one frame. The actual return value is a cons `(`*nimages* `.` *delay*`)`, where *nimages* is the number of frames and *delay* is the delay in seconds between them, or `nil` if the image does not specify a delay. Images that are intended to be animated usually specify a frame delay, whereas ones that are intended to be treated as multiple pages do not.

`image-current-frame` *image* [Function]
This function returns the index of the current frame number for *image*, counting from 0.

`image-show-frame` *image n* **&optional** *nocheck* [Function]
This function switches *image* to frame number *n*. It replaces a frame number outside the valid range with that of the end of the range, unless *nocheck* is non-`nil`. If *image* does not contain a frame with the specified number, the image displays as a hollow box.

`image-animate` *image* **&optional** *index limit* [Function]
This function animates *image*. The optional integer *index* specifies the frame from which to start (default 0). The optional argument *limit* controls the length of the animation. If omitted or `nil`, the image animates once only; if `t` it loops forever; if a number animation stops after that many seconds.

Animation operates by means of a timer. Note that Emacs imposes a minimum frame delay of 0.01 (`image-minimum-frame-delay`) seconds. If the image itself does not specify a delay, Emacs uses `image-default-frame-delay`.

`image-animate-timer` *image* [Function]
This function returns the timer responsible for animating *image*, if there is one.

41.17.11 Image Cache

Emacs caches images so that it can display them again more efficiently. When Emacs displays an image, it searches the image cache for an existing image specification `equal` to the desired specification. If a match is found, the image is displayed from the cache. Otherwise, Emacs loads the image normally.

`image-flush` *spec* **&optional** *frame* [Function]
This function removes the image with specification *spec* from the image cache of frame *frame*. Image specifications are compared using `equal`. If *frame* is `nil`, it defaults to the selected frame. If *frame* is `t`, the image is flushed on all existing frames.

In Emacs's current implementation, each graphical terminal possesses an image cache, which is shared by all the frames on that terminal (see Section 30.2 [Multiple Terminals], page 789). Thus, refreshing an image in one frame also refreshes it in all other frames on the same terminal.

One use for `image-flush` is to tell Emacs about a change in an image file. If an image specification contains a `:file` property, the image is cached based on the file's contents when the image is first displayed. Even if the file subsequently changes, Emacs continues displaying the old version of the image. Calling `image-flush` flushes the image from the cache, forcing Emacs to re-read the file the next time it needs to display that image.

Another use for `image-flush` is for memory conservation. If your Lisp program creates a large number of temporary images over a period much shorter than `image-cache-eviction-delay` (see below), you can opt to flush unused images yourself, instead of waiting for Emacs to do it automatically.

`clear-image-cache` **&optional** *filter* [Function]
This function clears an image cache, removing all the images stored in it. If *filter* is omitted or `nil`, it clears the cache for the selected frame. If *filter* is a frame, it clears the cache for that frame. If *filter* is `t`, all image caches are cleared. Otherwise, *filter* is taken to be a file name, and all images associated with that file name are removed from all image caches.

If an image in the image cache has not been displayed for a specified period of time, Emacs removes it from the cache and frees the associated memory.

`image-cache-eviction-delay` [Variable]
This variable specifies the number of seconds an image can remain in the cache without being displayed. When an image is not displayed for this length of time, Emacs removes it from the image cache.

Under some circumstances, if the number of images in the cache grows too large, the actual eviction delay may be shorter than this.

If the value is `nil`, Emacs does not remove images from the cache except when you explicitly clear it. This mode can be useful for debugging.

`image-cache-size` [Function]
: This function returns the total size of the current image cache, in bytes. An image of size 200x100 with 24 bits per color will have a cache size of 60000 bytes, for instance.

41.18 Icons

Emacs sometimes uses buttons (for clicking on) or small graphics (to illustrate something). Since Emacs is available on a wide variety of systems with different capabilities, and users have different preferences, Emacs provides a facility to handle this in a convenient way, allowing customization, graceful degradation, accessibility, as well as themability: *Icons*.

The central macro here is `define-icon`, and here's a simple example:

```
(define-icon outline-open button
  '((image "right.svg" "open.xpm" "open.pbm" :height line)
    (emoji "")
    (symbol "" "")
    (text "open" :face icon-button))
  "Icon used for buttons for opening a section in outline buffers."
  :version "29.1"
  :help-echo "Open this section")
```

Which alternative will actually be displayed depends on the value of the user option `icon-preference` (see Section "Icons" in *The GNU Emacs Manual*) and on the results of run-time checks for what the current frame's terminal can actually display.

The macro in the example above defines `outline-open` as an icon, and inherits properties from the icon called `button` (so this is meant as a clickable button to be inserted in a buffer). It is followed by a list of *icon types* along with the actual icon shapes themselves. In addition, there's a doc string and various keywords that contain additional information and properties.

To instantiate an icon, you use `icon-string`, which will consult the current Customize theming, and the `icon-preference` user option, and finally what the Emacs is able to actually display. If `icon-preference` is `(image emoji symbol text)` (i.e., allowing all of these forms of icons), in this case, `icon-string` will first check that Emacs is able to display images at all, and then whether it has support for each of those different image formats. If that fails, Emacs will check whether Emacs can display emojis (in the current frame). If that fails, it'll check whether it can display the symbol in question. If that fails, it'll use the plain text version.

For instance, if `icon-preference` doesn't contain `image` or `emoji`, it'll skip those entries.

Code can confidently call `icon-string` in all circumstances and be sure that something readable will appear on the screen, no matter whether the user is on a graphical terminal or a text terminal, and no matter which features Emacs was built with.

`define-icon` *name parent specs doc* **&rest** *keywords* [Macro]
: Define an icon *name*, a symbol, with the display alternatives in *spec*, that can be later instantiated using `icon-string`. The *name* is the name of the resulting keyword.

  The resulting icon will inherit specs from *parent*, and from their parent's parents, and so on, and the lowest descendent element wins.

specs is a list of icon specifications. The first element of each specification is the type, and the rest is something that can be used as an icon of that type, and then optionally followed by a keyword list. The following icon types are available:

`image` In this case, there may be many images listed as candidates. Emacs will choose the first one that the current Emacs instance can show. If an image is listed is an absolute file name, it's used as is, but it's otherwise looked up in the list `image-load-path` (see Section 41.17.8 [Defining Images], page 1225).

`emoji` This should be a (possibly colorful) emoji.

`symbol` This should be a (monochrome) symbol character.

`text` Icons should also have a textual fallback. This can also be used for the visually impaired: if `icon-preference` is just `(text)`, all icons will be replaced by text.

Various keywords may follow the list of icon specifications. For instance:

```
(symbol "" "" :face icon-button)
```

Unknown keywords are ignored. The following keywords are allowed:

`:face` The face to be used for the icon.

`:height` This is only valid for `image` icons, and can be either a number (which specifies the height in pixels), or the symbol `line`, which will use the default line height in the currently selected window.

`:width` This is only valid for `image` icons, and can be either a number (which specifies the width in pixels), or the symbol `font`, which will use the width in pixels of the current buffer's default face font.

doc should be a doc string.

keywords is a list of keyword/value pairs. The following keywords are allowed:

`:version` The (approximate) Emacs version this button first appeared. (This keyword is mandatory.)

`:group` The customization group this icon belongs in. If not present, it is inferred.

`:help-echo`
The help string shown when hovering over the icon with the mouse pointer.

`icon-string` *icon* [Function]
This function returns a string suitable for display in the current buffer for *icon*.

`icon-elements` *icon* [Function]
Alternatively, you can get a "deconstructed" version of *icon* with this function. It returns a plist (see Section 5.9 [Property Lists], page 99) where the keys are `string`, `face` and *image*. (The latter is only present if the icon is represented by an image.) This can be useful if the icon isn't to be inserted directly in the buffer, but needs some sort of pre-processing first.

Icons can be customized with *M-x customize-icon*. Themes can specify changes to icons with, for instance:

```
(custom-theme-set-icons
  'my-theme
  '(outline-open ((image :height 100)
                  (text " OPEN ")))
  '(outline-close ((image :height 100)
                   (text " CLOSE " :face warning))))
```

41.19 Embedded Native Widgets

Emacs is able to display native widgets, such as GTK+ WebKit widgets, in Emacs buffers when it was built with the necessary support libraries and is running on a graphical terminal. To test whether Emacs supports display of embedded widgets, check that the `xwidget-internal` feature is available (see Section 16.7 [Named Features], page 310).

To display an embedded widget in a buffer, you must first create an xwidget object, and then use that object as the display specifier in a `display` text or overlay property (see Section 41.16 [Display Property], page 1203).

Embedded widgets can send events notifying Lisp code about changes occurring within them. (see Section 22.7.11 [Xwidget Events], page 450).

`make-xwidget` *type title width height arguments* **&optional** *buffer related* [Function]

This creates and returns an xwidget object. If *buffer* is omitted or `nil`, it defaults to the current buffer. If *buffer* names a buffer that doesn't exist, it will be created. The *type* identifies the type of the xwidget component, it can be one of the following:

`webkit` The WebKit component.

The *width* and *height* arguments specify the widget size in pixels, and *title*, a string, specifies its title. *related* is used internally by the WebKit widget, and specifies another WebKit widget that the newly created widget should share settings and subprocesses with.

The xwidget that is returned will be killed alongside its buffer (see Section 28.10 [Killing Buffers], page 687). You can also kill it using `kill-xwidget`. Once it is killed, the xwidget may continue to exist as a Lisp object and act as a `display` property until all references to it are gone, but most actions that can be performed on live xwidgets will no longer be available.

`xwidgetp` *object* [Function]

This function returns `t` if *object* is an xwidget, `nil` otherwise.

`xwidget-live-p` *object* [Function]

This function returns `t` if *object* is an xwidget that hasn't been killed, and `nil` otherwise.

`kill-xwidget` *xwidget* [Function]

This function kills *xwidget*, by removing it from its buffer and releasing window system resources it holds.

`xwidget-plist` *xwidget* [Function]
This function returns the property list of *xwidget*.

`set-xwidget-plist` *xwidget plist* [Function]
This function replaces the property list of *xwidget* with a new property list given by *plist*.

`xwidget-buffer` *xwidget* [Function]
This function returns the buffer of *xwidget*. If *xwidget* has been killed, it returns `nil`.

`set-xwidget-buffer` *xwidget buffer* [Function]
This function sets the buffer of *xwidget* to *buffer*.

`get-buffer-xwidgets` *buffer* [Function]
This function returns a list of xwidget objects associated with the *buffer*, which can be specified as a buffer object or a name of an existing buffer, a string. The value is `nil` if *buffer* contains no xwidgets.

`xwidget-webkit-goto-uri` *xwidget uri* [Function]
This function browses the specified *uri* in the given *xwidget*. The *uri* is a string that specifies the name of a file or a URL.

`xwidget-webkit-execute-script` *xwidget script* [Function]
This function causes the browser widget specified by *xwidget* to execute the specified JavaScript `script`.

`xwidget-webkit-execute-script-rv` *xwidget script* **&optional** *default* [Function]
This function executes the specified *script* like `xwidget-webkit-execute-script` does, but it also returns the script's return value as a string. If *script* doesn't return a value, this function returns *default*, or `nil` if *default* was omitted.

`xwidget-webkit-get-title` *xwidget* [Function]
This function returns the title of *xwidget* as a string.

`xwidget-resize` *xwidget width height* [Function]
This function resizes the specified *xwidget* to the size *width*x*height* pixels.

`xwidget-size-request` *xwidget* [Function]
This function returns the desired size of *xwidget* as a list of the form `(`*`width height`*`)`. The dimensions are in pixels.

`xwidget-info` *xwidget* [Function]
This function returns the attributes of *xwidget* as a vector of the form `[`*`type title width height`*`]`. The attributes are usually determined by `make-xwidget` when the xwidget is created.

`set-xwidget-query-on-exit-flag` *xwidget flag* [Function]
This function allows you to arrange that Emacs will ask the user for confirmation before exiting or before killing a buffer that has *xwidget* associated with it. If *flag* is non-`nil`, Emacs will query the user, otherwise it will not.

`xwidget-query-on-exit-flag` *xwidget* [Function]
: This function returns the current setting of *xwidget*s query-on-exit flag, either `t` or `nil`.

`xwidget-perform-lispy-event` *xwidget event frame* [Function]
: Send an input event *event* to *xwidget*. The precise action performed is platform-specific. See Section 22.7 [Input Events], page 441.

  You can optionally pass the frame on which the event was generated via *frame*. On X11, modifier keys in key events will not be considered if *frame* is `nil`, and the selected frame is not an X-Windows frame.

  On GTK, only keyboard and function key events are supported. Mouse, motion, and click events are dispatched to the xwidget without going through Lisp code, and as such shouldn't require this function to be called.

`xwidget-webkit-search` *query xwidget* **&optional** *case-insensitive backwards wrap-around* [Function]
: Start an incremental search on the WebKit widget *xwidget* with the string *query* as the query. *case-insensitive* denotes whether or not the search is case-insensitive, *backwards* determines if the search is performed backwards towards the start of the document, and *wrap-around* determines whether or not the search terminates at the end of the document.

  If the function is called while a search query is already present, then the query specified here will replace the existing query.

  To stop a search query, use `xwidget-webkit-finish-search`.

`xwidget-webkit-next-result` *xwidget* [Function]
: Display the next search result in *xwidget*. This function will signal an error if a search query has not been already started in *xwidget* through `xwidget-webkit-search`.

  If `wrap-around` was non-`nil` when `xwidget-webkit-search` was called, then the search will restart from the beginning of the document when its end is reached.

`xwidget-webkit-previous-result` *xwidget* [Function]
: Display the previous search result in *xwidget*. This function signals an error if a search query has not been already started in *xwidget* through `xwidget-webkit-search`.

  If `wrap-around` was non-`nil` when `xwidget-webkit-search` was called, then the search will restart from the end of the document when its beginning is reached.

`xwidget-webkit-finish-search` *xwidget* [Function]
: Finish a search operation started with `xwidget-webkit-search` in *xwidget*. If there is no query currently ongoing, this function signals an error.

`xwidget-webkit-load-html` *xwidget text* **&optional** *base-uri* [Function]
: Load *text*, a string, into *xwidget*, which should be a WebKit xwidget. Any HTML markup in *text* will be processed by *xwidget* while rendering the text.

  Optional argument *base-uri*, which should be a string, specifies the absolute location of the web resources referenced by *text*, to be used for resolving relative links in *text*.

`xwidget-webkit-goto-history` *xwidget rel-pos* [Function]
Make *xwidget*, a WebKit widget, load the *rel-pos*th element in its navigation history.

If *rel-pos* is zero, the current page will be reloaded instead.

`xwidget-webkit-back-forward-list` *xwidget* **&optional** *limit* [Function]
Return the navigation history of *xwidget*, up to *limit* items in each direction. If not specified, *limit* defaults to 50.

The returned value is a list of the form `(back here forward)`, where *here* is the current navigation item, while *back* is a list of items containing the items recorded by WebKit before the current navigation item, and *forward* is a list of items recorded after the current navigation item. *back*, *here* and *forward* can all be `nil`.

When *here* is `nil`, it means that no items have been recorded yet; if *back* or *forward* are `nil`, it means that there is no history recorded before or after the current item respectively.

Navigation items are themselves lists of the form `(idx title uri)`. In these lists, *idx* is an index that can be passed to `xwidget-webkit-goto-history`, *title* is the human-readable title of the item, and *uri* is the URI of the item. The user should normally have no reason to load *uri* manually to reach a specific history item. Instead, *idx* should be passed as an index to `xwidget-webkit-goto-history`.

`xwidget-webkit-estimated-load-progress` *xwidget* [Function]
Return an estimate of how much data is remaining to be transferred before the page displayed by the WebKit widget *xwidget* is fully loaded.

The value returned is a float ranging between 0.0 and 1.0.

`xwidget-webkit-set-cookie-storage-file` *xwidget file* [Function]
Make the WebKit widget *xwidget* store cookies in *file*.

file must be an absolute file name. The new setting will also affect any xwidget that was created with *xwidget* as the `related` argument to `make-xwidget`, and widgets related to those as well.

If this function is not called at least once on *xwidget* or a related widget, *xwidget* will not store cookies on disk at all.

`xwidget-webkit-stop-loading` *xwidget* [Function]
Terminate any data transfer still in progress in the WebKit widget *xwidget* as part of a page-loading operation. If a page is not being loaded, this function does nothing.

41.20 Buttons

The Button package defines functions for inserting and manipulating *buttons* that can be activated with the mouse or via keyboard commands. These buttons are typically used for various kinds of hyperlinks.

A button is essentially a set of text or overlay properties, attached to a stretch of text in a buffer. These properties are called *button properties*. One of these properties, the *action property*, specifies a function which is called when the user invokes the button using the keyboard or the mouse. The action function may examine the button and use its other properties as desired.

In some ways, the Button package duplicates the functionality in the Widget package. See Section "Introduction" in *The Emacs Widget Library*. The advantage of the Button package is that it is faster, smaller, and simpler to program. From the point of view of the user, the interfaces produced by the two packages are very similar.

41.20.1 Button Properties

Each button has an associated list of properties defining its appearance and behavior, and other arbitrary properties may be used for application specific purposes. The following properties have special meaning to the Button package:

`action` The function to call when the user invokes the button, which is passed the single argument *button*. By default this is `ignore`, which does nothing.

`mouse-action`
This is similar to `action`, and when present, will be used instead of `action` for button invocations resulting from mouse-clicks (instead of the user hitting `RET`). If not present, mouse-clicks use `action` instead.

`face` This is an Emacs face controlling how buttons of this type are displayed; by default this is the `button` face.

`mouse-face`
This is an additional face which controls appearance during mouse-overs (merged with the usual button face); by default this is the usual Emacs `highlight` face.

`keymap` The button's keymap, defining bindings active within the button region. By default this is the usual button region keymap, stored in the variable `button-map`, which defines `RET` and `mouse-2` to invoke the button.

`type` The button type. See Section 41.20.2 [Button Types], page 1238.

`help-echo`
A string displayed by the Emacs tooltip help system; by default, `"mouse-2, RET: Push this button"`. Alternatively, a function that returns, or a form that evaluates to, a string to be displayed or `nil`. For details see [Text help-echo], page 928.

The function is called with three arguments, *window*, *object*, and *pos*. The second argument, *object*, is either the overlay that had the property (for overlay buttons), or the buffer containing the button (for text property buttons). The other arguments have the same meaning as for the special text property `help-echo`.

`follow-link`
The `follow-link` property, defining how a `mouse-1` click behaves on this button, See Section 33.19.8 [Clickable Text], page 937.

`button` All buttons have a non-`nil` `button` property, which may be useful in finding regions of text that comprise buttons (which is what the standard button functions do).

There are other properties defined for the regions of text in a button, but these are not generally interesting for typical uses.

41.20.2 Button Types

Every button has a *button type*, which defines default values for the button's properties. Button types are arranged in a hierarchy, with specialized types inheriting from more general types, so that it's easy to define special-purpose types of buttons for specific tasks.

`define-button-type` *name* **&rest** *properties* [Function]

Define a button type called *name* (a symbol). The remaining arguments form a sequence of *property value* pairs, specifying default property values for buttons with this type (a button's type may be set by giving it a `type` property when creating the button, using the `:type` keyword argument).

In addition, the keyword argument `:supertype` may be used to specify a button-type from which *name* inherits its default property values. Note that this inheritance happens only when *name* is defined; subsequent changes to a supertype are not reflected in its subtypes.

Using `define-button-type` to define default properties for buttons is not necessary—buttons without any specified type use the built-in button-type `button`—but it is encouraged, since doing so usually makes the resulting code clearer and more efficient.

41.20.3 Making Buttons

Buttons are associated with a region of text, using an overlay or text properties to hold button-specific information, all of which are initialized from the button's type (which defaults to the built-in button type `button`). Like all Emacs text, the appearance of the button is governed by the `face` property; by default (via the `face` property inherited from the `button` button-type) this is a simple underline, like a typical web-page link.

For convenience, there are two sorts of button-creation functions, those that add button properties to an existing region of a buffer, called `make-...button`, and those that also insert the button text, called `insert-...button`.

The button-creation functions all take the `&rest` argument *properties*, which should be a sequence of *property value* pairs, specifying properties to add to the button; see Section 41.20.1 [Button Properties], page 1237. In addition, the keyword argument `:type` may be used to specify a button-type from which to inherit other properties; see Section 41.20.2 [Button Types], page 1238. Any properties not explicitly specified during creation will be inherited from the button's type (if the type defines such a property).

The following functions add a button using an overlay (see Section 41.9 [Overlays], page 1154) to hold the button properties:

`make-button` *beg end* **&rest** *properties* [Function]

This makes a button from *beg* to *end* in the current buffer, and returns it.

`insert-button` *label* **&rest** *properties* [Function]

This inserts a button with the label *label* at point, and returns it.

The following functions are similar, but using text properties (see Section 33.19 [Text Properties], page 919) to hold the button properties. Such buttons do not add markers to the buffer, so editing in the buffer does not slow down if there is an extremely large numbers of buttons. However, if there is an existing face text property on the text (e.g.,

a face assigned by Font Lock mode), the button face may not be visible. Both of these functions return the starting position of the new button.

`make-text-button` *beg end* **&rest** *properties* [Function]
This makes a button from *beg* to *end* in the current buffer, using text properties.

`insert-text-button` *label* **&rest** *properties* [Function]
This inserts a button with the label *label* at point, using text properties.

`buttonize` *string callback* **&optional** *data* [Function]
Sometimes it's more convenient to make a string into a button without inserting it into a buffer immediately, for instance when creating data structures that may then, later, be inserted into a buffer. This function makes *string* into such a string, and *callback* will be called when the user clicks on the button. The optional *data* parameter will be used as the parameter when *callback* is called. If `nil`, the button is used as the parameter instead.

41.20.4 Manipulating Buttons

These are functions for getting and setting properties of buttons. Often these are used by a button's invocation function to determine what to do.

Where a *button* parameter is specified, it means an object referring to a specific button, either an overlay (for overlay buttons), or a buffer-position or marker (for text property buttons). Such an object is passed as the first argument to a button's invocation function when it is invoked.

`button-start` *button* [Function]
Return the position at which *button* starts.

`button-end` *button* [Function]
Return the position at which *button* ends.

`button-get` *button prop* [Function]
Get the property of button *button* named *prop*.

`button-put` *button prop val* [Function]
Set *button*'s *prop* property to *val*.

`button-activate` *button* **&optional** *use-mouse-action* [Function]
Call *button*'s `action` property (i.e., invoke the function that is the value of that property, passing it the single argument *button*). If *use-mouse-action* is non-`nil`, try to invoke the button's `mouse-action` property instead of `action`; if the button has no `mouse-action` property, use `action` as normal. If the `button-data` property is present in *button*, use that as the argument for the `action` function instead of *button*.

`button-label` *button* [Function]
Return *button*'s text label.

`button-type` *button* [Function]
Return *button*'s button-type.

`button-has-type-p` *button type* [Function]
Return `t` if *button* has button-type *type*, or one of *type*'s subtypes.

`button-at` *pos* [Function]
Return the button at position *pos* in the current buffer, or `nil`. If the button at *pos* is a text property button, the return value is a marker pointing to *pos*.

`button-type-put` *type prop val* [Function]
Set the button-type *type*'s *prop* property to *val*.

`button-type-get` *type prop* [Function]
Get the property of button-type *type* named *prop*.

`button-type-subtype-p` *type supertype* [Function]
Return `t` if button-type *type* is a subtype of *supertype*.

41.20.5 Button Buffer Commands

These are commands and functions for locating and operating on buttons in an Emacs buffer.

`push-button` is the command that a user uses to actually push a button, and is bound by default in the button itself to `RET` and to `mouse-2` using a local keymap in the button's overlay or text properties. Commands that are useful outside the buttons itself, such as `forward-button` and `backward-button` are additionally available in the keymap stored in `button-buffer-map`; a mode which uses buttons may want to use `button-buffer-map` as a parent keymap for its keymap. Alternatively, the `button-mode` can be switched on for much the same effect: It's a minor mode that does nothing else than install `button-buffer-map` as a minor mode keymap.

If the button has a non-`nil` `follow-link` property, and `mouse-1-click-follows-link` is set, a quick `mouse-1` click will also activate the `push-button` command. See Section 33.19.8 [Clickable Text], page 937.

`push-button` **&optional** *pos use-mouse-action* [Command]
Perform the action specified by a button at location *pos*. *pos* may be either a buffer position or a mouse-event. If *use-mouse-action* is non-`nil`, or *pos* is a mouse-event (see Section 22.7.3 [Mouse Events], page 443), try to invoke the button's `mouse-action` property instead of `action`; if the button has no `mouse-action` property, use `action` as normal. *pos* defaults to point, except when `push-button` is invoked interactively as the result of a mouse-event, in which case, the mouse event's position is used. If there's no button at *pos*, do nothing and return `nil`, otherwise return `t`.

`forward-button` *n* **&optional** *wrap display-message no-error* [Command]
Move to the *n*th next button, or *n*th previous button if *n* is negative. If *n* is zero, move to the start of any button at point. If *wrap* is non-`nil`, moving past either end of the buffer continues from the other end. If *display-message* is non-`nil`, the button's help-echo string is displayed. Any button with a non-`nil` `skip` property is skipped over. Returns the button found, and signals an error if no buttons can be found. If *no-error* is non-`nil`, return `nil` instead of signaling the error.

`backward-button` *n* **&optional** *wrap display-message no-error* [Command]
: Move to the *n*th previous button, or *n*th next button if *n* is negative. If *n* is zero, move to the start of any button at point. If *wrap* is non-`nil`, moving past either end of the buffer continues from the other end. If *display-message* is non-`nil`, the button's help-echo string is displayed. Any button with a non-`nil` `skip` property is skipped over. Returns the button found, and signals an error if no buttons can be found. If *no-error* is non-`nil`, return `nil` instead of signaling the error.

`next-button` *pos* **&optional** *count-current* [Function]
`previous-button` *pos* **&optional** *count-current* [Function]
: Return the next button after (for `next-button`) or before (for `previous-button`) position *pos* in the current buffer. If *count-current* is non-`nil`, count any button at *pos* in the search, instead of starting at the next button.

41.21 Abstract Display

The Ewoc package constructs buffer text that represents a structure of Lisp objects, and updates the text to follow changes in that structure. This is like the "view" component in the "model–view–controller" design paradigm. Ewoc means "Emacs's Widget for Object Collections".

An *ewoc* is a structure that organizes information required to construct buffer text that represents certain Lisp data. The buffer text of the ewoc has three parts, in order: first, fixed *header* text; next, textual descriptions of a series of data elements (Lisp objects that you specify); and last, fixed *footer* text. Specifically, an ewoc contains information on:

- The buffer which its text is generated in.
- The text's start position in the buffer.
- The header and footer strings.
- A doubly-linked chain of *nodes*, each of which contains:
  - A *data element*, a single Lisp object.
  - Links to the preceding and following nodes in the chain.
- A *pretty-printer* function which is responsible for inserting the textual representation of a data element value into the current buffer.

Typically, you define an ewoc with `ewoc-create`, and then pass the resulting ewoc structure to other functions in the Ewoc package to build nodes within it, and display it in the buffer. Once it is displayed in the buffer, other functions determine the correspondence between buffer positions and nodes, move point from one node's textual representation to another, and so forth. See Section 41.21.1 [Abstract Display Functions], page 1242.

A node *encapsulates* a data element much the way a variable holds a value. Normally, encapsulation occurs as a part of adding a node to the ewoc. You can retrieve the data element value and place a new value in its place, like so:

```
(ewoc-data node)
⇒ value

(ewoc-set-data node new-value)
⇒ new-value
```

You can also use, as the data element value, a Lisp object (list or vector) that is a container for the real value, or an index into some other structure. The example (see Section 41.21.2 [Abstract Display Example], page 1244) uses the latter approach.

When the data changes, you will want to update the text in the buffer. You can update all nodes by calling `ewoc-refresh`, or just specific nodes using `ewoc-invalidate`, or all nodes satisfying a predicate using `ewoc-map`. Alternatively, you can delete invalid nodes using `ewoc-delete` or `ewoc-filter`, and add new nodes in their place. Deleting a node from an ewoc deletes its associated textual description from buffer, as well.

41.21.1 Abstract Display Functions

In this subsection, *ewoc* and *node* stand for the structures described above (see Section 41.21 [Abstract Display], page 1241), while *data* stands for an arbitrary Lisp object used as a data element.

`ewoc-create` *pretty-printer* **&optional** *header footer nosep* [Function]
: This constructs and returns a new ewoc, with no nodes (and thus no data elements). *pretty-printer* should be a function that takes one argument, a data element of the sort you plan to use in this ewoc, and inserts its textual description at point using `insert` (and never `insert-before-markers`, because that would interfere with the Ewoc package's internal mechanisms).

  Normally, a newline is automatically inserted after the header, the footer and every node's textual description. If *nosep* is non-`nil`, no newline is inserted. This may be useful for displaying an entire ewoc on a single line, for example, or for making nodes invisible by arranging for *pretty-printer* to do nothing for those nodes.

  An ewoc maintains its text in the buffer that is current when you create it, so switch to the intended buffer before calling `ewoc-create`.

`ewoc-buffer` *ewoc* [Function]
: This returns the buffer where *ewoc* maintains its text.

`ewoc-get-hf` *ewoc* [Function]
: This returns a cons cell (`header` . `footer`) made from *ewoc*'s header and footer.

`ewoc-set-hf` *ewoc header footer* [Function]
: This sets the header and footer of *ewoc* to the strings *header* and *footer*, respectively.

`ewoc-enter-first` *ewoc data* [Function]
`ewoc-enter-last` *ewoc data* [Function]
: These add a new node encapsulating *data*, putting it, respectively, at the beginning or end of *ewoc*'s chain of nodes.

`ewoc-enter-before` *ewoc node data* [Function]
`ewoc-enter-after` *ewoc node data* [Function]
: These add a new node encapsulating *data*, adding it to *ewoc* before or after *node*, respectively.

`ewoc-prev` *ewoc node* [Function]
`ewoc-next` *ewoc node* [Function]
: These return, respectively, the previous node and the next node of *node* in *ewoc*.

`ewoc-nth` *ewoc n* [Function]
This returns the node in *ewoc* found at zero-based index *n*. A negative *n* means count from the end. `ewoc-nth` returns `nil` if *n* is out of range.

`ewoc-data` *node* [Function]
This extracts the data encapsulated by *node* and returns it.

`ewoc-set-data` *node data* [Function]
This sets the data encapsulated by *node* to *data*.

`ewoc-locate` *ewoc* **&optional** *pos guess* [Function]
This determines the node in *ewoc* which contains point (or *pos* if specified), and returns that node. If *ewoc* has no nodes, it returns `nil`. If *pos* is before the first node, it returns the first node; if *pos* is after the last node, it returns the last node. The optional third arg *guess* should be a node that is likely to be near *pos*; this doesn't alter the result, but makes the function run faster.

`ewoc-location` *node* [Function]
This returns the start position of *node*.

`ewoc-goto-prev` *ewoc arg* [Function]
`ewoc-goto-next` *ewoc arg* [Function]
These move point to the previous or next, respectively, *arg*th node in *ewoc*. `ewoc-goto-prev` does not move if it is already at the first node or if *ewoc* is empty, whereas `ewoc-goto-next` moves past the last node, returning `nil`. Excepting this special case, these functions return the node moved to.

`ewoc-goto-node` *ewoc node* [Function]
This moves point to the start of *node* in *ewoc*.

`ewoc-refresh` *ewoc* [Function]
This function regenerates the text of *ewoc*. It works by deleting the text between the header and the footer, i.e., all the data elements' representations, and then calling the pretty-printer function for each node, one by one, in order.

`ewoc-invalidate` *ewoc* **&rest** *nodes* [Function]
This is similar to `ewoc-refresh`, except that only *nodes* in *ewoc* are updated instead of the entire set.

`ewoc-delete` *ewoc* **&rest** *nodes* [Function]
This deletes each node in *nodes* from *ewoc*.

`ewoc-filter` *ewoc predicate* **&rest** *args* [Function]
This calls *predicate* for each data element in *ewoc* and deletes those nodes for which *predicate* returns `nil`. Any *args* are passed to *predicate*.

`ewoc-collect` *ewoc predicate* **&rest** *args* [Function]
This calls *predicate* for each data element in *ewoc* and returns a list of those elements for which *predicate* returns non-`nil`. The elements in the list are ordered as in the buffer. Any *args* are passed to *predicate*.

`ewoc-map` *map-function ewoc* **&rest** *args* [Function]
This calls *map-function* for each data element in *ewoc* and updates those nodes for which *map-function* returns non-`nil`. Any *args* are passed to *map-function*.

41.21.2 Abstract Display Example

Here is a simple example using functions of the ewoc package to implement a *color components* display, an area in a buffer that represents a vector of three integers (itself representing a 24-bit RGB value) in various ways.

```
(setq colorcomp-ewoc nil
      colorcomp-data nil
      colorcomp-mode-map nil
      colorcomp-labels ["Red" "Green" "Blue"])

(defun colorcomp-pp (data)
  (if data
      (let ((comp (aref colorcomp-data data)))
        (insert (aref colorcomp-labels data) "\t: #x"
                (format "%02X" comp) " "
                (make-string (ash comp -2) ?#) "\n"))
    (let ((cstr (format "#%02X%02X%02X"
                        (aref colorcomp-data 0)
                        (aref colorcomp-data 1)
                        (aref colorcomp-data 2)))
          (samp " (sample text) "))
      (insert "Color\t: "
              (propertize samp 'face
                          `(foreground-color . ,cstr))
              (propertize samp 'face
                          `(background-color . ,cstr))
              "\n"))))

(defun colorcomp (color)
  "Allow fiddling with COLOR in a new buffer.
The buffer is in Color Components mode."
  (interactive "sColor (name or #RGB or #RRGGBB): ")
  (when (string= "" color)
    (setq color "green"))
  (unless (color-values color)
    (error "No such color: %S" color))
  (switch-to-buffer
   (generate-new-buffer (format "originally: %s" color)))
  (kill-all-local-variables)
  (setq major-mode 'colorcomp-mode
        mode-name "Color Components")
  (use-local-map colorcomp-mode-map)
  (erase-buffer)
```

```
  (buffer-disable-undo)
  (let ((data (apply 'vector (mapcar (lambda (n) (ash n -8))
                                     (color-values color))))
        (ewoc (ewoc-create 'colorcomp-pp
                           "\nColor Components\n\n"
                           (substitute-command-keys
                            "\n\\{colorcomp-mode-map}"))))
    (set (make-local-variable 'colorcomp-data) data)
    (set (make-local-variable 'colorcomp-ewoc) ewoc)
    (ewoc-enter-last ewoc 0)
    (ewoc-enter-last ewoc 1)
    (ewoc-enter-last ewoc 2)
    (ewoc-enter-last ewoc nil)))
```

This example can be extended to be a color selection widget (in other words, the "controller" part of the "model–view–controller" design paradigm) by defining commands to modify `colorcomp-data` and to finish the selection process, and a keymap to tie it all together conveniently.

```
(defun colorcomp-mod (index limit delta)
  (let ((cur (aref colorcomp-data index)))
    (unless (= limit cur)
      (aset colorcomp-data index (+ cur delta)))
    (ewoc-invalidate
     colorcomp-ewoc
     (ewoc-nth colorcomp-ewoc index)
     (ewoc-nth colorcomp-ewoc -1))))

(defun colorcomp-R-more () (interactive) (colorcomp-mod 0 255 1))
(defun colorcomp-G-more () (interactive) (colorcomp-mod 1 255 1))
(defun colorcomp-B-more () (interactive) (colorcomp-mod 2 255 1))
(defun colorcomp-R-less () (interactive) (colorcomp-mod 0 0 -1))
(defun colorcomp-G-less () (interactive) (colorcomp-mod 1 0 -1))
(defun colorcomp-B-less () (interactive) (colorcomp-mod 2 0 -1))

(defun colorcomp-copy-as-kill-and-exit ()
  "Copy the color components into the kill ring and kill the buffer.
The string is formatted #RRGGBB (hash followed by six hex digits)."
  (interactive)
  (kill-new (format "#%02X%02X%02X"
                    (aref colorcomp-data 0)
                    (aref colorcomp-data 1)
                    (aref colorcomp-data 2)))
  (kill-buffer nil))

(setq colorcomp-mode-map
      (define-keymap :suppress t
        "i" 'colorcomp-R-less
        "o" 'colorcomp-R-more
        "k" 'colorcomp-G-less
        "l" 'colorcomp-G-more
        "," 'colorcomp-B-less
        "." 'colorcomp-B-more
        "SPC" 'colorcomp-copy-as-kill-and-exit))
```

Note that we never modify the data in each node, which is fixed when the ewoc is created to be either `nil` or an index into the vector `colorcomp-data`, the actual color components.

41.22 Blinking Parentheses

This section describes the mechanism by which Emacs shows a matching open parenthesis when the user inserts a close parenthesis.

`blink-paren-function` [Variable]
: The value of this variable should be a function (of no arguments) to be called whenever a character with close parenthesis syntax is inserted. The value of `blink-paren-function` may be `nil`, in which case nothing is done.

`blink-matching-paren` [User Option]
: If this variable is `nil`, then `blink-matching-open` does nothing.

`blink-matching-paren-distance` [User Option]
: This variable specifies the maximum distance to scan for a matching parenthesis before giving up.

`blink-matching-delay` [User Option]
: This variable specifies the number of seconds to keep indicating the matching parenthesis. A fraction of a second often gives good results, but the default is 1, which works on all systems.

`blink-matching-open` [Command]
: This function is the default value of `blink-paren-function`. It assumes that point follows a character with close parenthesis syntax and applies the appropriate effect momentarily to the matching opening character. If that character is not already on the screen, it displays the character's context in the echo area. To avoid long delays, this function does not search farther than `blink-matching-paren-distance` characters.

  Here is an example of calling this function explicitly.

```
(defun interactive-blink-matching-open ()
  "Indicate momentarily the start of parenthesized sexp before point."
  (interactive)
  (let ((blink-matching-paren-distance
         (buffer-size))
        (blink-matching-paren t))
    (blink-matching-open)))
```

41.23 Character Display

This section describes how characters are actually displayed by Emacs. Typically, a character is displayed as a *glyph* (a graphical symbol which occupies one character position on the screen), whose appearance corresponds to the character itself. For example, the character '`a`' (character code 97) is displayed as '`a`'. Some characters, however, are displayed specially. For example, the formfeed character (character code 12) is usually displayed as a sequence of two glyphs, '`^L`', while the newline character (character code 10) starts a new screen line.

You can modify how each character is displayed by defining a *display table*, which maps each character code into a sequence of glyphs. See Section 41.23.2 [Display Tables], page 1248.

41.23.1 Usual Display Conventions

Here are the conventions for displaying each character code (in the absence of a display table, which can override these conventions).

- The *printable ASCII characters*, character codes 32 through 126 (consisting of numerals, English letters, and symbols like '`#`') are displayed literally.
- The tab character (character code 9) displays as whitespace stretching up to the next tab stop column. See Section "Text Display" in *The GNU Emacs Manual*. The variable `tab-width` controls the number of spaces per tab stop (see below).
- The newline character (character code 10) has a special effect: it ends the preceding line and starts a new line.
- The non-printable *ASCII control characters*—character codes 0 through 31, as well as the `DEL` character (character code 127)—display in one of two ways according to the variable `ctl-arrow`. If this variable is non-`nil` (the default), these characters are displayed as sequences of two glyphs, where the first glyph is '`^`' (a display table can specify a glyph to use instead of '`^`'); e.g., the `DEL` character is displayed as '`^?`'.

  If `ctl-arrow` is `nil`, these characters are displayed as octal escapes (see below).

  This rule also applies to carriage return (character code 13), if that character appears in the buffer. But carriage returns usually do not appear in buffer text; they are eliminated as part of end-of-line conversion (see Section 34.10.1 [Coding System Basics], page 980).
- *Raw bytes* are non-ASCII characters with codes 128 through 255 (see Section 34.1 [Text Representations], page 966). These characters display as *octal escapes*: sequences of four glyphs, where the first glyph is the ASCII code for '\', and the others are digit characters representing the character code in octal. (A display table can specify a glyph to use instead of '\'.)
- Each non-ASCII character with code above 255 is displayed literally, if the terminal supports it. If the terminal does not support it, the character is said to be *glyphless*, and it is usually displayed using a placeholder glyph. For example, if a graphical terminal has no font for a character, Emacs usually displays a box containing the character code in hexadecimal. See Section 41.23.5 [Glyphless Chars], page 1250.

The above display conventions apply even when there is a display table, for any character whose entry in the active display table is `nil`. Thus, when you set up a display table, you need only specify the characters for which you want special display behavior.

The following variables affect how certain characters are displayed on the screen. Since they change the number of columns the characters occupy, they also affect the indentation functions. They also affect how the mode line is displayed; if you want to force redisplay of the mode line using the new values, call the function `force-mode-line-update` (see Section 24.4 [Mode Line Format], page 551).

`ctl-arrow` [User Option]
: This buffer-local variable controls how control characters are displayed. If it is non-`nil`, they are displayed as a caret followed by the character: '`^A`'. If it is `nil`, they are displayed as octal escapes: a backslash followed by three octal digits, as in '`\001`'.

`tab-width` [User Option]
: The value of this buffer-local variable is the spacing between tab stops used for displaying tab characters in Emacs buffers. The value is in units of columns, and the default is 8. Note that this feature is completely independent of the user-settable tab stops used by the command `tab-to-tab-stop`. See Section 33.17.5 [Indent Tabs], page 917.

41.23.2 Display Tables

A display table is a special-purpose char-table (see Section 6.6 [Char-Tables], page 119), with `display-table` as its subtype, which is used to override the usual character display conventions. This section describes how to make, inspect, and assign elements to a display table object. The next section (see Section 41.23.3 [Active Display Table], page 1249) describes the various standard display tables and their precedence.

`make-display-table` [Function]
: This creates and returns a display table. The table initially has `nil` in all elements.

The ordinary elements of the display table are indexed by character codes; the element at index *c* says how to display the character code *c*. The value should be `nil` (which means to display the character *c* according to the usual display conventions; see Section 41.23.1 [Usual Display], page 1247), or a vector of glyph codes (which means to display the character *c* as those glyphs; see Section 41.23.4 [Glyphs], page 1250).

Warning: if you use the display table to change the display of newline characters, the whole buffer will be displayed as one long line.

The display table also has six *extra slots* which serve special purposes. Here is a table of their meanings; `nil` in any slot means to use the default for that slot, as stated below.

0
: The glyph for the end of a truncated screen line (the default for this is '`$`'). See Section 41.23.4 [Glyphs], page 1250. On graphical terminals, Emacs by default uses arrows in the fringes to indicate truncation, so the display table has no effect, unless you disable the fringes (see Section "Window Fringes" in *the GNU Emacs Manual*).

1
: The glyph for the end of a continued line (the default is '\'). On graphical terminals, Emacs by default uses curved arrows in the fringes to indicate continuation, so the display table has no effect, unless you disable the fringes.

2
: The glyph for indicating a character displayed as an octal character code (the default is '\').

3
: The glyph for indicating a control character (the default is '^').

4
: A vector of glyphs for indicating the presence of invisible lines (the default is '...'). See Section 41.7 [Selective Display], page 1149.

5 The glyph used to draw the border between side-by-side windows (the default is '|'). See Section 29.7 [Splitting Windows], page 710. This currently has effect only on text terminals; on graphical terminals, if vertical scroll bars are supported and in use, a scroll bar separates the two windows, and if there are no vertical scroll bars and no dividers (see Section 41.15 [Window Dividers], page 1202), Emacs uses a thin line to indicate the border.

For example, here is how to construct a display table that mimics the effect of setting `ctl-arrow` to a non-`nil` value (see Section 41.23.4 [Glyphs], page 1250, for the function `make-glyph-code`):

```
(setq disptab (make-display-table))
(dotimes (i 32)
  (or (= i ?\t)
      (= i ?\n)
      (aset disptab i
            (vector (make-glyph-code ?^ 'escape-glyph)
                    (make-glyph-code (+ i 64) 'escape-glyph)))))
(aset disptab 127
      (vector (make-glyph-code ?^ 'escape-glyph)
              (make-glyph-code ?? 'escape-glyph)))
```

`display-table-slot` *display-table slot* [Function]
This function returns the value of the extra slot *slot* of *display-table*. The argument *slot* may be a number from 0 to 5 inclusive, or a slot name (symbol). Valid symbols are `truncation`, `wrap`, `escape`, `control`, `selective-display`, and `vertical-border`.

`set-display-table-slot` *display-table slot value* [Function]
This function stores *value* in the extra slot *slot* of *display-table*. The argument *slot* may be a number from 0 to 5 inclusive, or a slot name (symbol). Valid symbols are `truncation`, `wrap`, `escape`, `control`, `selective-display`, and `vertical-border`.

`describe-display-table` *display-table* [Function]
This function displays a description of the display table *display-table* in a help buffer.

`describe-current-display-table` [Command]
This command displays a description of the current display table in a help buffer.

41.23.3 Active Display Table

Each window can specify a display table, and so can each buffer. The window's display table, if there is one, takes precedence over the buffer's display table. If neither exists, Emacs tries to use the standard display table; if that is `nil`, Emacs uses the usual character display conventions (see Section 41.23.1 [Usual Display], page 1247). (Emacs does not "merge" display tables: For instance, if the window has a display table, the buffer's display table and the standard display table are completely ignored.)

Note that display tables affect how the mode line is displayed, so if you want to force redisplay of the mode line using a new display table, call `force-mode-line-update` (see Section 24.4 [Mode Line Format], page 551).

`window-display-table` **&optional** *window* [Function]
This function returns *window*'s display table, or `nil` if there is none. The default for *window* is the selected window.

`set-window-display-table` *window table* [Function]
This function sets the display table of *window* to *table*. The argument *table* should be either a display table or `nil`.

`buffer-display-table` [Variable]
This variable is automatically buffer-local in all buffers; its value specifies the buffer's display table. If it is `nil`, there is no buffer display table.

`standard-display-table` [Variable]
The value of this variable is the standard display table, which is used when Emacs is displaying a buffer in a window with neither a window display table nor a buffer display table defined, or when Emacs is outputting text to the standard output or error streams. Although its default is typically `nil`, in an interactive session if the terminal cannot display curved quotes, its default maps curved quotes to ASCII approximations. See Section 25.4 [Text Quoting Style], page 602.

The `disp-table` library defines several functions for changing the standard display table.

41.23.4 Glyphs

A *glyph* is a graphical symbol which occupies a single character position on the screen. Each glyph is represented in Lisp as a *glyph code*, which specifies a character and optionally a face to display it in (see Section 41.12 [Faces], page 1168). The main use of glyph codes is as the entries of display tables (see Section 41.23.2 [Display Tables], page 1248). The following functions are used to manipulate glyph codes:

`make-glyph-code` *char* **&optional** *face* [Function]
This function returns a glyph code representing char *char* with face *face*. If *face* is omitted or `nil`, the glyph uses the default face; in that case, the glyph code is an integer. If *face* is non-`nil`, the glyph code is not necessarily an integer object.

`glyph-char` *glyph* [Function]
This function returns the character of glyph code *glyph*.

`glyph-face` *glyph* [Function]
This function returns face of glyph code *glyph*, or `nil` if *glyph* uses the default face.

41.23.5 Glyphless Character Display

Glyphless characters are characters which are displayed in a special way, e.g., as a box containing a hexadecimal code, instead of being displayed literally. These include characters which are explicitly defined to be glyphless, as well as characters for which there is no available font (on a graphical display), and characters which cannot be encoded by the terminal's coding system (on a text terminal).

The `glyphless-display-mode` minor mode can be used to toggle displaying glyphless characters in a convenient manner in the current buffer. If this mode is enabled, all the glyphless characters are displayed as boxes that display acronyms of their character names.

`glyphless-char-display` [Variable]

For more fine-grained (and global) control, this variable can be used. The value of this variable is a char-table which defines glyphless characters and how they are displayed. Each entry must be one of the following display methods:

`nil` Display the character in the usual way.

`zero-width`
: Don't display the character.

`thin-space`
: Display a thin space, 1-pixel wide on graphical displays, or 1-character wide on text terminals.

`empty-box`
: Display an empty box.

`hex-code` Display a box containing the Unicode codepoint of the character, in hexadecimal notation.

an ASCII string
: Display a box containing that string. The string should contain at most 6 ASCII characters. As an exception, if the string includes just one character, on text-mode terminals that character will be displayed without a box; this allows to handle such "acronyms" as a replacement character for characters that cannot be displayed by the terminal.

a cons cell `(`*`graphical . text`*`)`
: Display with *graphical* on graphical displays, and with *text* on text terminals. Both *graphical* and *text* must be one of the display methods described above.

The `thin-space`, `empty-box`, `hex-code`, and ASCII string display methods are drawn with the `glyphless-char` face. On text terminals, a box is emulated by square brackets, '`[]`'.

The char-table has one extra slot, which determines how to display any character that cannot be displayed with any available font, or cannot be encoded by the terminal's coding system. Its value should be one of the above display methods, except `zero-width`.

If a character has a non-`nil` entry in an active display table, the display table takes effect; in this case, Emacs does not consult `glyphless-char-display` at all.

`glyphless-char-display-control` [User Option]

This user option provides a convenient way to set `glyphless-char-display` for groups of similar characters. Do not set its value directly from Lisp code; the value takes effect only via a custom `:set` function (see Section 15.3 [Variable Definitions], page 281), which updates `glyphless-char-display`.

Its value should be an alist of elements `(`*`group . method`*`)`, where *group* is a symbol specifying a group of characters, and *method* is a symbol specifying how to display them.

group should be one of the following:

`c0-control`
: ASCII control characters `U+0000` to `U+001F`, excluding the newline and tab characters (normally displayed as escape sequences like '`^A`'; see Section "How Text Is Displayed" in *The GNU Emacs Manual*).

`c1-control`
: Non-ASCII, non-printing characters `U+0080` to `U+009F` (normally displayed as octal escape sequences like '`\230`').

`format-control`
: Characters of Unicode General Category [Cf], such as U+200E LEFT-TO-RIGHT MARK, but excluding characters that have graphic images, such as U+00AD SOFT HYPHEN.

`bidi-control`
: This is a subset of `format-control`, but only includes characters that are related to bidirectional formatting control, like U+2069 POP DIRECTIONAL ISOLATE and U+202A LEFT-TO-RIGHT EMBEDDING. See Section 41.27 [Bidirectional Display], page 1255.

  Characters of Unicode General Category [Cf], such as U+200E LEFT-TO-RIGHT MARK, but excluding characters that have graphic images, such as U+00AD SOFT HYPHEN.

`variation-selectors`
: Unicode VS-1 through VS-256 (U+FE00 through U+FE0F and U+E0100 through U+E01EF), which are used to select between different glyphs for the same codepoints (typically emojis).

`no-font`
: Characters for which there is no suitable font, or which cannot be encoded by the terminal's coding system, or those for which the text-mode terminal has no glyphs.

The *method* symbol should be one of `zero-width`, `thin-space`, `empty-box`, or `hex-code`. These have the same meanings as in `glyphless-char-display`, above.

41.24 Beeping

This section describes how to make Emacs ring the bell (or blink the screen) to attract the user's attention. Be conservative about how often you do this; frequent bells can become irritating. Also be careful not to use just beeping when signaling an error is more appropriate (see Section 11.7.3 [Errors], page 179).

`ding` **&optional** *do-not-terminate* [Function]
: This function beeps, or flashes the screen (see `visible-bell` below). It also terminates any keyboard macro currently executing unless *do-not-terminate* is non-`nil`.

`beep` **&optional** *do-not-terminate* [Function]
: This is a synonym for `ding`.

`visible-bell` [User Option]
: This variable determines whether Emacs should flash the screen to represent a bell. Non-`nil` means yes, `nil` means no. This is effective on graphical displays, and on text terminals provided the terminal's Termcap entry defines the visible bell capability (‘`vb`’).

`ring-bell-function` [User Option]
: If this is non-`nil`, it specifies how Emacs should ring the bell. Its value should be a function of no arguments. If this is non-`nil`, it takes precedence over the `visible-bell` variable.

41.25 Window Systems

Emacs works with several window systems, most notably the X Window System. Both Emacs and X use the term “window”, but use it differently. An Emacs frame is a single window as far as X is concerned; the individual Emacs windows are not known to X at all.

`window-system` [Variable]
: This terminal-local variable tells Lisp programs what window system Emacs is using for displaying the frame. The possible values are

  `x` Emacs is displaying the frame using X.

  `w32` Emacs is displaying the frame using native MS-Windows GUI.

  `ns` Emacs is displaying the frame using the Nextstep interface (used on GNUstep and macOS).

  `pc` Emacs is displaying the frame using MS-DOS direct screen writes.

  `haiku` Emacs is displaying the frame using the Application Kit on Haiku.

  `pgtk` Emacs is displaying the frame using pure GTK facilities.

  `nil` Emacs is displaying the frame on a character-based terminal.

`initial-window-system` [Variable]
: This variable holds the value of `window-system` used for the first frame created by Emacs during startup. (When Emacs is invoked as a daemon, it does not create any initial frames, so `initial-window-system` is `nil`, except on MS-Windows, where it is still `w32`. See Section “Initial Options” in *The GNU Emacs Manual*.)

`window-system` **&optional** *frame* [Function]
: This function returns a symbol whose name tells what window system is used for displaying *frame* (which defaults to the currently selected frame). The list of possible symbols it returns is the same one documented for the variable `window-system` above.

Do *not* use `window-system` and `initial-window-system` as predicates or boolean flag variables, if you want to write code that works differently on text terminals and graphic displays. That is because `window-system` is not a good indicator of Emacs capabilities on a given display type. Instead, use `display-graphic-p` or any of the other `display-*-p` predicates described in Section 30.26 [Display Feature Testing], page 852.

41.26 Tooltips

Tooltips are special frames (see Chapter 30 [Frames], page 787) that are used to display helpful hints (a.k.a. "tips") related to the current position of the mouse pointer. Emacs uses tooltips to display help strings about active portions of text (see Section 33.19.4 [Special Properties], page 927) and about various UI elements, such as menu items (see Section 23.18.1.2 [Extended Menu Items], page 512) and tool-bar buttons (see Section 23.18.6 [Tool Bar], page 518).

`tooltip-mode` [Function]

> Tooltip Mode is a minor mode that enables display of tooltips. Turning off this mode causes the tooltips be displayed in the echo area. On text-mode (a.k.a. "TTY") frames, tooltips are always displayed in the echo area.

When Emacs is built with the GTK+ toolkit or Haiku windowing support, it by default displays tooltips using toolkit functions, and the appearance of the tooltips is then controlled by the toolkit's settings. Toolkit-provided tooltips can be disabled by changing the value of the variable `use-system-tooltips` to `nil`. The rest of this subsection describes how to control non-toolkit tooltips, which are presented by Emacs itself.

Tooltips are displayed in special frames called tooltip frames, which have their own frame parameters (see Section 30.4 [Frame Parameters], page 804). Unlike other frames, the default parameters for tooltip frames are stored in a special variable.

`tooltip-frame-parameters` [User Option]

> This customizable option holds the default frame parameters used for displaying tooltips. Any font and color parameters are ignored, and the corresponding attributes of the `tooltip` face are used instead. If `left` or `top` parameters are included, they are used as absolute frame-relative coordinates where the tooltip should be shown. (Mouse-relative position of the tooltip can be customized using the variables described in Section "Tooltips" in *The GNU Emacs Manual*.) Note that the `left` and `top` parameters, if present, override the values of mouse-relative offsets.

The `tooltip` face determines the appearance of text shown in tooltips. It should generally use a variable-pitch font of size that is preferably smaller than the default frame font.

`tooltip-functions` [Variable]

> This abnormal hook is a list of functions to call when Emacs needs to display a tooltip. Each function is called with a single argument *event* which is a copy of the last mouse movement event. If a function on this list actually displays the tooltip, it should return non-`nil`, and then the rest of the functions will not be called. The default value of this variable is a single function `tooltip-help-tips`.

If you write your own function to be put on the `tooltip-functions` list, you may need to know the buffer of the mouse event that triggered the tooltip display. The following function provides that information.

`tooltip-event-buffer` *event* [Function]

> This function returns the buffer over which *event* occurred. Call it with the argument of the function from `tooltip-functions` to obtain the buffer whose text triggered

the tooltip. Note that the event might occur not over a buffer (e.g., over the tool bar), in which case this function will return `nil`.

Other aspects of tooltip display are controlled by several customizable settings; see Section "Tooltips" in *The GNU Emacs Manual*.

41.27 Bidirectional Display

Emacs can display text written in scripts, such as Arabic, Farsi, and Hebrew, whose natural ordering for horizontal text display runs from right to left. Furthermore, segments of Latin script and digits embedded in right-to-left text are displayed left-to-right, while segments of right-to-left script embedded in left-to-right text (e.g., Arabic or Hebrew text in comments or strings in a program source file) are appropriately displayed right-to-left. We call such mixtures of left-to-right and right-to-left text *bidirectional text*. This section describes the facilities and options for editing and displaying bidirectional text.

Text is stored in Emacs buffers and strings in *logical* (or *reading*) order, i.e., the order in which a human would read each character. In right-to-left and bidirectional text, the order in which characters are displayed on the screen (called *visual order*) is not the same as logical order; the characters' screen positions do not increase monotonically with string or buffer position. In performing this *bidirectional reordering*, Emacs follows the Unicode Bidirectional Algorithm (a.k.a. UBA), which is described in Annex #9 of the Unicode standard (`https://www.unicode.org/reports/tr9/`). Emacs provides a "Full Bidirectionality" class implementation of the UBA, consistent with the requirements of the Unicode Standard v9.0. Note, however, that the way Emacs displays continuation lines when text direction is opposite to the base paragraph direction deviates from the UBA, which requires to perform line wrapping before reordering text for display.

`bidi-display-reordering` [Variable]

If the value of this buffer-local variable is non-`nil` (the default), Emacs performs bidirectional reordering for display. The reordering affects buffer text, as well as display strings and overlay strings from text and overlay properties in the buffer (see Section 41.9.2 [Overlay Properties], page 1157, and see Section 41.16 [Display Property], page 1203). If the value is `nil`, Emacs does not perform bidirectional reordering in the buffer.

The default value of `bidi-display-reordering` controls the reordering of strings which are not directly supplied by a buffer, including the text displayed in mode lines (see Section 24.4 [Mode Line Format], page 551) and header lines (see Section 24.4.7 [Header Lines], page 560).

Emacs never reorders the text of a unibyte buffer, even if `bidi-display-reordering` is non-`nil` in the buffer. This is because unibyte buffers contain raw bytes, not characters, and thus lack the directionality properties required for reordering. Therefore, to test whether text in a buffer will be reordered for display, it is not enough to test the value of `bidi-display-reordering` alone. The correct test is this:

```
(if (and enable-multibyte-characters
         bidi-display-reordering)
    ;; Buffer is being reordered for display
  )
```

However, unibyte display and overlay strings *are* reordered if their parent buffer is reordered. This is because plain-ASCII strings are stored by Emacs as unibyte strings. If a unibyte display or overlay string includes non-ASCII characters, these characters are assumed to have left-to-right direction.

Text covered by `display` text properties, by overlays with `display` properties whose value is a string, and by any other properties that replace buffer text, is treated as a single unit when it is reordered for display. That is, the entire chunk of text covered by these properties is reordered together. Moreover, the bidirectional properties of the characters in such a chunk of text are ignored, and Emacs reorders them as if they were replaced with a single character `U+FFFC`, known as the *Object Replacement Character*. This means that placing a display property over a portion of text may change the way that the surrounding text is reordered for display. To prevent this unexpected effect, always place such properties on text whose directionality is identical with text that surrounds it.

Each paragraph of bidirectional text has a *base direction*, either right-to-left or left-to-right. Left-to-right paragraphs are displayed beginning at the left margin of the window, and are truncated or continued when the text reaches the right margin. Right-to-left paragraphs are displayed beginning at the right margin, and are continued or truncated at the left margin.

Where exactly paragraphs start and end, for the purpose of the Emacs UBA implementation, is determined by the following two buffer-local variables (note that `paragraph-start` and `paragraph-separate` have no influence on this). By default both of these variables are `nil`, and paragraphs are bounded by empty lines, i.e., lines that consist entirely of zero or more whitespace characters followed by a newline.

`bidi-paragraph-start-re` [Variable]
: If non-`nil`, this variable's value should be a regular expression matching a line that starts or separates two paragraphs. The regular expression is always matched after a newline, so it is best to anchor it, i.e., begin it with a `"^"`.

`bidi-paragraph-separate-re` [Variable]
: If non-`nil`, this variable's value should be a regular expression matching a line separates two paragraphs. The regular expression is always matched after a newline, so it is best to anchor it, i.e., begin it with a `"^"`.

If you modify any of these two variables, you should normally modify both, to make sure they describe paragraphs consistently. For example, to have each new line start a new paragraph for bidi-reordering purposes, set both variables to `"^"`.

By default, Emacs determines the base direction of each paragraph by looking at the text at its beginning. The precise method of determining the base direction is specified by the UBA; in a nutshell, the first character in a paragraph that has an explicit directionality determines the base direction of the paragraph. However, sometimes a buffer may need to force a certain base direction for its paragraphs. For example, buffers containing program source code should force all paragraphs to be displayed left-to-right. You can use following variable to do this:

`bidi-paragraph-direction` [User Option]
: If the value of this buffer-local variable is the symbol `right-to-left` or `left-to-right`, all paragraphs in the buffer are assumed to have that specified direction. Any

other value is equivalent to `nil` (the default), which means to determine the base direction of each paragraph from its contents.

Modes for program source code should set this to `left-to-right`. Prog mode does this by default, so modes derived from Prog mode do not need to set this explicitly (see Section 24.2.5 [Basic Major Modes], page 537).

`current-bidi-paragraph-direction` **&optional** *buffer* [Function]
This function returns the paragraph direction at point in the named *buffer*. The returned value is a symbol, either `left-to-right` or `right-to-left`. If *buffer* is omitted or `nil`, it defaults to the current buffer. If the buffer-local value of the variable `bidi-paragraph-direction` is non-`nil`, the returned value will be identical to that value; otherwise, the returned value reflects the paragraph direction determined dynamically by Emacs. For buffers whose value of `bidi-display-reordering` is `nil` as well as unibyte buffers, this function always returns `left-to-right`.

Sometimes there's a need to move point in strict visual order, either to the left or to the right of its current screen position. Emacs provides a primitive to do that.

`move-point-visually` *direction* [Function]
This function moves point of the currently selected window to the buffer position that appears immediately to the right or to the left of point on the screen. If *direction* is positive, point will move one screen position to the right, otherwise it will move one screen position to the left. Note that, depending on the surrounding bidirectional context, this could potentially move point many buffer positions away. If invoked at the end of a screen line, the function moves point to the rightmost or leftmost screen position of the next or previous screen line, as appropriate for the value of *direction*.

The function returns the new buffer position as its value.

Bidirectional reordering can have surprising and unpleasant effects when two strings with bidirectional content are juxtaposed in a buffer, or otherwise programmatically concatenated into a string of text. A typical problematic case is when a buffer consists of sequences of text fields separated by whitespace or punctuation characters, like Buffer Menu mode or Rmail Summary Mode. Because the punctuation characters used as separators have *weak directionality*, they take on the directionality of surrounding text. As result, a numeric field that follows a field with bidirectional content can be displayed *to the left* of the preceding field, messing up the expected layout. There are several ways to avoid this problem:

- Append the special character U+200E LEFT-TO-RIGHT MARK, or LRM, to the end of each field that may have bidirectional content, or prepend it to the beginning of the following field. The function `bidi-string-mark-left-to-right`, described below, comes in handy for this purpose. (In a right-to-left paragraph, use U+200F RIGHT-TO-LEFT MARK, or RLM, instead.) This is one of the solutions recommended by the UBA.
- Include the tab character in the field separator. The tab character plays the role of *segment separator* in bidirectional reordering, causing the text on either side to be reordered separately.
- Separate fields with a `display` property or overlay with a property value of the form `(space . PROPS)` (see Section 41.16.2 [Specified Space], page 1205). Emacs treats this

display specification as a *paragraph separator*, and reorders the text on either side separately.

`bidi-string-mark-left-to-right` *string* [Function]
: This function returns its argument *string*, possibly modified, such that the result can be safely concatenated with another string, or juxtaposed with another string in a buffer, without disrupting the relative layout of this string and the next one on display. If the string returned by this function is displayed as part of a left-to-right paragraph, it will always appear on display to the left of the text that follows it. The function works by examining the characters of its argument, and if any of those characters could cause reordering on display, the function appends the LRM character to the string. The appended LRM character is made invisible by giving it an `invisible` text property of `t` (see Section 41.6 [Invisible Text], page 1146).

The reordering algorithm uses the bidirectional properties of the characters stored as their `bidi-class` property (see Section 34.6 [Character Properties], page 971). Lisp programs can change these properties by calling the `put-char-code-property` function. However, doing this requires a thorough understanding of the UBA, and is therefore not recommended. Any changes to the bidirectional properties of a character have global effect: they affect all Emacs frames and windows.

Similarly, the `mirroring` property is used to display the appropriate mirrored character in the reordered text. Lisp programs can affect the mirrored display by changing this property. Again, any such changes affect all of Emacs display.

The bidirectional properties of characters can be overridden by inserting into the text special directional control characters, LEFT-TO-RIGHT OVERRIDE (LRO) and RIGHT-TO-LEFT OVERRIDE (RLO). Any characters between a RLO and the following newline or POP DIRECTIONAL FORMATTING (PDF) control character, whichever comes first, will be displayed as if they were strong right-to-left characters, i.e. they will be reversed on display. Similarly, any characters between LRO and PDF or newline will display as if they were strong left-to-right, and will *not* be reversed even if they are strong right-to-left characters.

These overrides are useful when you want to make some text unaffected by the reordering algorithm, and instead directly control the display order. But they can also be used for malicious purposes, known as *phishing*. Specifically, a URL on a Web page or a link in an email message can be manipulated to make its visual appearance unrecognizable, or similar to some popular benign location, while the real location, interpreted by a browser in the logical order, is very different.

Emacs provides a primitive that applications can use to detect instances of text whose bidirectional properties were overridden so as to make a left-to-right character display as if it were a right-to-left character, or vice versa.

`bidi-find-overridden-directionality` *from to* **&optional** *object* [Function]
: This function looks at the text of the specified *object* between positions *from* (inclusive) and *to* (exclusive), and returns the first position where it finds a strong left-to-right character whose directional properties were forced to display the character as right-to-left, or for a strong right-to-left character that was forced to display as

left-to-right. If it finds no such characters in the specified region of text, it returns `nil`.

The optional argument *object* specifies which text to search, and defaults to the current buffer. If *object* is non-`nil`, it can be some other buffer, or it can be a string or a window. If it is a string, the function searches that string. If it is a window, the function searches the buffer displayed in that window. If a buffer whose text you want to examine is displayed in some window, we recommend to specify it by that window, rather than pass the buffer to the function. This is because telling the function about the window allows it to correctly account for window-specific overlays, which might change the result of the function if some text in the buffer is covered by overlays.

When text that includes mixed right-to-left and left-to-right characters and bidirectional controls is copied into a different location, it can change its visual appearance, and also can affect the visual appearance of the surrounding text at destination. This is because reordering of bidirectional text specified by the UBA has non-trivial context-dependent effects both on the copied text and on the text at copy destination that will surround it.

Sometimes, a Lisp program may need to preserve the exact visual appearance of the copied text at destination, and of the text that surrounds the copy. Lisp programs can use the following function to achieve that effect.

`buffer-substring-with-bidi-context` *start end* **&optional** *no-properties* [Function]

This function works similar to `buffer-substring` (see Section 33.2 [Buffer Contents], page 881), but it prepends and appends to the copied text bidi directional control characters necessary to preserve the visual appearance of the text when it is inserted at another place. Optional argument *no-properties*, if non-`nil`, means remove the text properties from the copy of the text.

42 Operating System Interface

This chapter is about starting and getting out of Emacs, access to values in the operating system environment, and terminal input, output.

See Section E.1 [Building Emacs], page 1351, for related information. See Chapter 41 [Display], page 1133, for additional operating system status information pertaining to the terminal and the screen.

42.1 Starting Up Emacs

This section describes what Emacs does when it is started, and how you can customize these actions.

42.1.1 Summary: Sequence of Actions at Startup

When Emacs is started up, it performs the following operations (see `normal-top-level` in `startup.el`):

1. It adds subdirectories to `load-path`, by running the file named `subdirs.el` in each directory in the list. Normally, this file adds the directory's subdirectories to the list, and those are scanned in their turn. The files `subdirs.el` are normally generated automatically when Emacs is installed.
2. It loads any `leim-list.el` that it finds in the `load-path` directories. This file is intended for registering input methods. The search is only for any personal `leim-list.el` files that you may have created; it skips the directories containing the standard Emacs libraries (these should contain only a single `leim-list.el` file, which is compiled into the Emacs executable).
3. It sets the variable `before-init-time` to the value of `current-time` (see Section 42.5 [Time of Day], page 1275). It also sets `after-init-time` to `nil`, which signals to Lisp programs that Emacs is being initialized.
4. It sets the language environment and the terminal coding system, if requested by environment variables such as `LANG`.
5. It does some basic parsing of the command-line arguments.
6. It loads your early init file (see Section "Early Init File" in *The GNU Emacs Manual*). This is not done if the options '`-q`', '`-Q`', or '`--batch`' were specified. If the '`-u`' option was specified, Emacs looks for the init file in that user's home directory instead.
7. It calls the function `package-activate-all` to activate any optional Emacs Lisp package that has been installed. See Section 43.1 [Packaging Basics], page 1308. However, Emacs doesn't activate the packages when `package-enable-at-startup` is `nil` or when it's started with one of the options '`-q`', '`-Q`', or '`--batch`'. To activate the packages in the latter case, `package-activate-all` should be called explicitly (e.g., via the '`--funcall`' option).
8. If not running in batch mode, it initializes the window system that the variable `initial-window-system` specifies (see Section 41.25 [Window Systems], page 1253). The initialization function, `window-system-initialization`, is a *generic function* (see Section 13.8 [Generic Functions], page 245) whose actual implementation is different for each supported window system. If the value of `initial-window-system`

is *windowsystem*, then the appropriate implementation of the initialization function is defined in the file `term/windowsystem-win.el`. This file should have been compiled into the Emacs executable when it was built.

9. It runs the normal hook `before-init-hook`.
10. If appropriate, it creates a graphical frame. As part of creating the graphical frame, it initializes the window system specified by `initial-frame-alist` and `default-frame-alist` (see Section 30.4.2 [Initial Parameters], page 806) for the graphical frame, by calling the `window-system-initialization` function for that window system. This is not done in batch (noninteractive) or daemon mode.
11. It initializes the initial frame's faces, and sets up the menu bar and tool bar if needed. If graphical frames are supported, it sets up the tool bar even if the current frame is not a graphical one, since a graphical frame may be created later on.
12. It use `custom-reevaluate-setting` to re-initialize the members of the list `custom-delayed-init-variables`. These are any pre-loaded user options whose default value depends on the run-time, rather than build-time, context. See Section E.1 [Building Emacs], page 1351.
13. It loads the library `site-start`, if it exists. This is not done if the options '`-Q`' or '`--no-site-file`' were specified.
14. It loads your init file (see Section 42.1.2 [Init File], page 1263). This is not done if the options '`-q`', '`-Q`', or '`--batch`' were specified. If the '`-u`' option was specified, Emacs looks for the init file in that user's home directory instead.
15. It loads the library `default`, if it exists. This is not done if `inhibit-default-init` is non-`nil`, nor if the options '`-q`', '`-Q`', or '`--batch`' were specified.
16. It loads your abbrevs from the file specified by `abbrev-file-name`, if that file exists and can be read (see Section 38.3 [Abbrev Files], page 1071). This is not done if the option '`--batch`' was specified.
17. It sets the variable `after-init-time` to the value of `current-time`. This variable was set to `nil` earlier; setting it to the current time signals that the initialization phase is over, and, together with `before-init-time`, provides the measurement of how long it took.
18. It runs the normal hooks `after-init-hook` and `delayed-warnings-hook`. The latter shows any warnings emitted during previous stages of startup, which are automatically delayed.
19. If the buffer `*scratch*` exists and is still in Fundamental mode (as it should be by default), it sets its major mode according to `initial-major-mode`.
20. If started on a text terminal, it loads the terminal-specific Lisp library (see Section 42.1.3 [Terminal-Specific], page 1265), and runs the hook `tty-setup-hook`. This is not done in `--batch` mode, nor if `term-file-prefix` is `nil`.
21. It displays the initial echo area message, unless you have suppressed that with `inhibit-startup-echo-area-message`.
22. It processes any command-line options that were not handled earlier.
23. It now exits if the option `--batch` was specified.
24. If the `*scratch*` buffer exists and is empty, it inserts `(substitute-command-keys initial-scratch-message)` into that buffer.

25. If `initial-buffer-choice` is a string, it visits the file (or directory) with that name. If it is a function, it calls the function with no arguments and selects the buffer that it returns. If one file is given as a command line argument, that file is visited and its buffer displayed alongside `initial-buffer-choice`. If more than one file is given, all of the files are visited and the `*Buffer List*` buffer is displayed alongside `initial-buffer-choice`.
26. It runs `emacs-startup-hook`.
27. It calls `frame-notice-user-settings`, which modifies the parameters of the selected frame according to whatever the init files specify.
28. It runs `window-setup-hook`. The only difference between this hook and `emacs-startup-hook` is that this one runs after the previously mentioned modifications to the frame parameters.
29. It displays the *startup screen*, which is a special buffer that contains information about copyleft and basic Emacs usage. This is not done if `inhibit-startup-screen` or `initial-buffer-choice` are non-`nil`, or if the '`--no-splash`' or '`-Q`' command-line options were specified.
30. If a daemon was requested, it calls `server-start`. (On POSIX systems, if a background daemon was requested, it then detaches from the controlling terminal.) See Section "Emacs Server" in *The GNU Emacs Manual*.
31. If started by the X session manager, it calls `emacs-session-restore` passing it as argument the ID of the previous session. See Section 42.18 [Session Management], page 1295.

The following options affect some aspects of the startup sequence.

`inhibit-startup-screen` [User Option]

This variable, if non-`nil`, inhibits the startup screen. In that case, Emacs typically displays the `*scratch*` buffer; but see `initial-buffer-choice`, below.

Do not set this variable in the init file of a new user, or in a way that affects more than one user, as that would prevent new users from receiving information about copyleft and basic Emacs usage.

`inhibit-startup-message` and `inhibit-splash-screen` are aliases for this variable.

`initial-buffer-choice` [User Option]

If non-`nil`, this variable is a string that specifies a file or directory for Emacs to display after starting up, instead of the startup screen. If its value is a function, Emacs calls that function which must return a buffer which is then displayed. If its value is `t`, Emacs displays the `*scratch*` buffer.

`inhibit-startup-echo-area-message` [User Option]

This variable controls the display of the startup echo area message. You can suppress the startup echo area message by adding text with this form to your init file:

```
(setq inhibit-startup-echo-area-message
      "your-login-name")
```

Emacs explicitly checks for an expression as shown above in your init file; your login name must appear in the expression as a Lisp string constant. You can also use

the Customize interface. Other methods of setting `inhibit-startup-echo-area-message` to the same value do not inhibit the startup message. This way, you can easily inhibit the message for yourself if you wish, but thoughtless copying of your init file will not inhibit the message for someone else.

`initial-scratch-message` [User Option]
This variable, if non-`nil`, should be a string, which is treated as documentation to be inserted into the `*scratch*` buffer when Emacs starts up or when that buffer is recreated. If it is `nil`, the `*scratch*` buffer is empty.

The following command-line options affect some aspects of the startup sequence. See Section "Initial Options" in *The GNU Emacs Manual*.

`--no-splash`
Do not display a splash screen.

`--batch` Run without an interactive terminal. See Section 42.17 [Batch Mode], page 1294.

`--daemon`
`--bg-daemon`
`--fg-daemon`
Do not initialize any display; just start a server. (A "background" daemon automatically runs in the background.)

`--no-init-file`
`-q` Do not load either the init file, or the `default` library.

`--no-site-file`
Do not load the `site-start` library.

`--quick`
`-Q` Equivalent to '`-q --no-site-file --no-splash`'.

`--init-directory`
Specify the directory to use when finding the Emacs init files.

42.1.2 The Init File

When you start Emacs, it normally attempts to load your *init file*. This is either a file named `.emacs` or `.emacs.el` in your home directory, or a file named `init.el` in a subdirectory named `.emacs.d` in your home directory.

The command-line switches '`-q`', '`-Q`', and '`-u`' control whether and where to find the init file; '`-q`' (and the stronger '`-Q`') says not to load an init file, while '`-u` ***user***' says to load *user*'s init file instead of yours. See Section "Entering Emacs" in *The GNU Emacs Manual*. If neither option is specified, Emacs uses the `LOGNAME` environment variable, or the `USER` (most systems) or `USERNAME` (MS systems) variable, to find your home directory and thus your init file; this way, even if you have su'd, Emacs still loads your own init file. If those environment variables are absent, though, Emacs uses your user-id to find your home directory.

Emacs also attempts to load a second init file, called the *early init file*, if it exists. This is a file named `early-init.el` in your `~/.emacs.d` directory. The difference between the

early init file and the regular init file is that the early init file is loaded much earlier during the startup process, so you can use it to customize some things that are initialized before loading the regular init file. For example, you can customize the process of initializing the package system, by setting variables such as *package-load-list* or *package-enable-at-startup*. See Section "Package Installation" in *The GNU Emacs Manual*.

An Emacs installation may have a *default init file*, which is a Lisp library named `default.el`. Emacs finds this file through the standard search path for libraries (see Section 16.1 [How Programs Do Loading], page 299). The Emacs distribution does not come with this file; it is intended for local customizations. If the default init file exists, it is loaded whenever you start Emacs. But your own personal init file, if any, is loaded first; if it sets `inhibit-default-init` to a non-`nil` value, then Emacs does not subsequently load the `default.el` file. In batch mode, or if you specify '`-q`' (or '`-Q`'), Emacs loads neither your personal init file nor the default init file.

Another file for site-customization is `site-start.el`. Emacs loads this *before* the user's init file. You can inhibit the loading of this file with the option '`--no-site-file`'.

`site-run-file` [User Option]
: This variable specifies the site-customization file to load before the user's init file. Its normal value is `"site-start"`. The only way you can change it with real effect is to do so before dumping Emacs.

See Section "Init File Examples" in *The GNU Emacs Manual*, for examples of how to make various commonly desired customizations in your `.emacs` file.

`inhibit-default-init` [User Option]
: If this variable is non-`nil`, it prevents Emacs from loading the default initialization library file. The default value is `nil`.

`before-init-hook` [Variable]
: This normal hook is run, once, just before loading all the init files (`site-start.el`, your init file, and `default.el`). (The only way to change it with real effect is before dumping Emacs.)

`after-init-hook` [Variable]
: This normal hook is run, once, just after loading all the init files (`site-start.el`, your init file, and `default.el`), before loading the terminal-specific library (if started on a text terminal) and processing the command-line action arguments.

`emacs-startup-hook` [Variable]
: This normal hook is run, once, just after handling the command line arguments. In batch mode, Emacs does not run this hook.

`window-setup-hook` [Variable]
: This normal hook is very similar to `emacs-startup-hook`. The only difference is that it runs slightly later, after setting of the frame parameters. See Section 42.1.1 [Startup Summary], page 1260.

`user-init-file` [Variable]
: This variable holds the absolute file name of the user's init file. If the actual init file loaded is a compiled file, such as `.emacs.elc`, the value refers to the corresponding source file.

`user-emacs-directory` [Variable]

This variable holds the name of the Emacs default directory. It defaults to `${XDG_CONFIG_HOME-'~/.config'}/emacs/` if that directory exists and `~/.emacs.d/` and `~/.emacs` do not exist, otherwise to `~/.emacs.d/` on all platforms but MS-DOS. Here, `${XDG_CONFIG_HOME-'~/.config'}` stands for the value of the environment variable `XDG_CONFIG_HOME` if that variable is set, and for `~/.config` otherwise. See Section "How Emacs Finds Your Init File" in *The GNU Emacs Manual*.

42.1.3 Terminal-Specific Initialization

Each terminal type can have its own Lisp library that Emacs loads when run on that type of terminal. The library's name is constructed by concatenating the value of the variable `term-file-prefix` and the terminal type (specified by the environment variable `TERM`). Normally, `term-file-prefix` has the value `"term/"`; changing this is not recommended. If there is an entry matching `TERM` in the `term-file-aliases` association list, Emacs uses the associated value in place of `TERM`. Emacs finds the file in the normal manner, by searching the `load-path` directories, and trying the '`.elc`' and '`.el`' suffixes.

The usual role of a terminal-specific library is to enable special keys to send sequences that Emacs can recognize. It may also need to set or add to `input-decode-map` if the Termcap or Terminfo entry does not specify all the terminal's function keys. See Section 42.13 [Terminal Input], page 1290.

When the name of the terminal type contains a hyphen or underscore, and no library is found whose name is identical to the terminal's name, Emacs strips from the terminal's name the last hyphen or underscore and everything that follows it, and tries again. This process is repeated until Emacs finds a matching library, or until there are no more hyphens or underscores in the name (i.e., there is no terminal-specific library). For example, if the terminal name is '`xterm-256color`' and there is no `term/xterm-256color.el` library, Emacs tries to load `term/xterm.el`. If necessary, the terminal library can evaluate `(getenv "TERM")` to find the full name of the terminal type.

Your init file can prevent the loading of the terminal-specific library by setting the variable `term-file-prefix` to `nil`.

You can also arrange to override some of the actions of the terminal-specific library by using `tty-setup-hook`. This is a normal hook that Emacs runs after initializing a new text terminal. You could use this hook to define initializations for terminals that do not have their own libraries. See Section 24.1 [Hooks], page 525.

`term-file-prefix` [User Option]

If the value of this variable is non-`nil`, Emacs loads a terminal-specific initialization file as follows:

```
(load (concat term-file-prefix (getenv "TERM")))
```

You may set the `term-file-prefix` variable to `nil` in your init file if you do not wish to load the terminal-initialization file.

On MS-DOS, Emacs sets the `TERM` environment variable to '`internal`'.

`term-file-aliases` [User Option]

This variable is an association list mapping terminal types to their aliases. For example, an element of the form `("vt102" . "vt100")` means to treat a terminal of type '`vt102`' like one of type '`vt100`'.

`tty-setup-hook` [Variable]

This variable is a normal hook that Emacs runs after initializing a new text terminal. (This applies when Emacs starts up in non-windowed mode, and when making a tty `emacsclient` connection.) The hook runs after loading your init file (if applicable) and the terminal-specific Lisp file, so you can use it to adjust the definitions made by that file.

For a related feature, see Section 42.1.2 [Init File], page 1263.

42.1.4 Command-Line Arguments

You can use command-line arguments to request various actions when you start Emacs. Note that the recommended way of using Emacs is to start it just once, after logging in, and then do all editing in the same Emacs session (see Section "Entering Emacs" in *The GNU Emacs Manual*). For this reason, you might not use command-line arguments very often; nonetheless, they can be useful when invoking Emacs from session scripts or debugging Emacs. This section describes how Emacs processes command-line arguments.

`command-line` [Function]

This function parses the command line that Emacs was called with, processes it, and (amongst other things) loads the user's init file and displays the startup messages.

`command-line-processed` [Variable]

The value of this variable is `t` once the command line has been processed.

If you redump Emacs by calling `dump-emacs` (see Section E.1 [Building Emacs], page 1351), you may wish to set this variable to `nil` first in order to cause the new dumped Emacs to process its new command-line arguments.

`command-switch-alist` [Variable]

This variable is an alist of user-defined command-line options and associated handler functions. By default it is empty, but you can add elements if you wish.

A *command-line option* is an argument on the command line, which has the form:

```
-option
```

The elements of the `command-switch-alist` look like this:

```
(option . handler-function)
```

The CAR, *option*, is a string, the name of a command-line option (including the initial hyphen). The *handler-function* is called to handle *option*, and receives the option name as its sole argument.

In some cases, the option is followed in the command line by an argument. In these cases, the *handler-function* can find all the remaining command-line arguments in the variable `command-line-args-left` (see below). (The entire list of command-line arguments is in `command-line-args`.)

Note that the handling of `command-switch-alist` doesn't treat equals signs in *option* specially. That is, if there's an option like `--name=value` on the command line, then only a `command-switch-alist` member whose `car` is literally `--name=value` will match this option. If you want to parse such options, you need to use `command-line-functions` instead (see below).

The command-line arguments are parsed by the `command-line-1` function in the `startup.el` file. See also Section "Command Line Arguments for Emacs Invocation" in *The GNU Emacs Manual*.

`command-line-args` [Variable]
The value of this variable is the list of command-line arguments passed to Emacs.

`command-line-args-left` [Variable]
The value of this variable is the list of command-line arguments that have not yet been processed.

`command-line-functions` [Variable]
This variable's value is a list of functions for handling an unrecognized command-line argument. Each time the next argument to be processed has no special meaning, the functions in this list are called, in order of appearance, until one of them returns a non-`nil` value.

These functions are called with no arguments. They can access the command-line argument under consideration through the variable `argi`, which is bound temporarily at this point. The remaining arguments (not including the current one) are in the variable `command-line-args-left`.

When a function recognizes and processes the argument in `argi`, it should return a non-`nil` value to say it has dealt with that argument. If it has also dealt with some of the following arguments, it can indicate that by deleting them from `command-line-args-left`.

If all of these functions return `nil`, then the argument is treated as a file name to visit.

42.2 Getting Out of Emacs

There are two ways to get out of Emacs: you can kill the Emacs job, which exits permanently, or you can suspend it, which permits you to reenter the Emacs process later. (In a graphical environment, you can of course simply switch to another application without doing anything special to Emacs, then switch back to Emacs when you want.)

42.2.1 Killing Emacs

Killing Emacs means ending the execution of the Emacs process. If you started Emacs from a terminal, the parent process normally resumes control. The low-level primitive for killing Emacs is `kill-emacs`.

`kill-emacs` **&optional** *exit-data restart* [Command]
This command calls the hook `kill-emacs-hook`, then exits the Emacs process and kills it.

If *exit-data* is an integer, that is used as the exit status of the Emacs process. (This is useful primarily in batch operation; see Section 42.17 [Batch Mode], page 1294.)

If *exit-data* is a string, its contents are stuffed into the terminal input buffer so that the shell (or whatever program next reads input) can read them.

If *exit-data* is neither an integer nor a string, or is omitted, that means to use the (system-specific) exit status which indicates successful program termination.

If *restart* is non-`nil`, instead of just exiting at the end, start a new Emacs process, using the same command line arguments as the currently running Emacs process.

The `kill-emacs` function is normally called via the higher-level command `C-x C-c` (`save-buffers-kill-terminal`). See Section "Exiting" in *The GNU Emacs Manual*. It is also called automatically if Emacs receives a `SIGTERM` or `SIGHUP` operating system signal (e.g., when the controlling terminal is disconnected), or if it receives a `SIGINT` signal while running in batch mode (see Section 42.17 [Batch Mode], page 1294).

`kill-emacs-hook` [Variable]

This normal hook is run by `kill-emacs`, before it kills Emacs.

Because `kill-emacs` can be called in situations where user interaction is impossible (e.g., when the terminal is disconnected), functions on this hook should not attempt to interact with the user. If you want to interact with the user when Emacs is shutting down, use `kill-emacs-query-functions`, described below.

When Emacs is killed, all the information in the Emacs process, aside from files that have been saved, is lost. Because killing Emacs inadvertently can lose a lot of work, the `save-buffers-kill-terminal` command queries for confirmation if you have buffers that need saving or subprocesses that are running. It also runs the abnormal hook `kill-emacs-query-functions`:

`kill-emacs-query-functions` [User Option]

When `save-buffers-kill-terminal` is killing Emacs, it calls the functions in this hook, after asking the standard questions and before calling `kill-emacs`. The functions are called in order of appearance, with no arguments. Each function can ask for additional confirmation from the user. If any of them returns `nil`, `save-buffers-kill-emacs` does not kill Emacs, and does not run the remaining functions in this hook. Calling `kill-emacs` directly does not run this hook.

`restart-emacs` [Command]

This command does the same as `save-buffers-kill-emacs`, but instead of just killing the current Emacs process at the end, it'll restart a new Emacs process, using the same command line arguments as the currently running Emacs process.

42.2.2 Suspending Emacs

On text terminals, it is possible to *suspend Emacs*, which means stopping Emacs temporarily and returning control to its superior process, which is usually the shell. This allows you to resume editing later in the same Emacs process, with the same buffers, the same kill ring, the same undo history, and so on. To resume Emacs, use the appropriate command in the parent shell—most likely `fg`.

Suspending works only on a terminal device from which the Emacs session was started. We call that device the *controlling terminal* of the session. Suspending is not allowed if the controlling terminal is a graphical terminal. Suspending is usually not relevant in graphical environments, since you can simply switch to another application without doing anything special to Emacs.

Some operating systems (those without `SIGTSTP`, or MS-DOS) do not support suspension of jobs; on these systems, suspension actually creates a new shell temporarily as a subprocess of Emacs. Then you would exit the shell to return to Emacs.

`suspend-emacs` **&optional** *string* [Command]

This function stops Emacs and returns control to the superior process. If and when the superior process resumes Emacs, `suspend-emacs` returns `nil` to its caller in Lisp.

This function works only on the controlling terminal of the Emacs session; to relinquish control of other tty devices, use `suspend-tty` (see below). If the Emacs session uses more than one terminal, you must delete the frames on all the other terminals before suspending Emacs, or this function signals an error. See Section 30.2 [Multiple Terminals], page 789.

If *string* is non-`nil`, its characters are sent to Emacs's superior shell, to be read as terminal input. The characters in *string* are not echoed by the superior shell; only the results appear.

Before suspending, `suspend-emacs` runs the normal hook `suspend-hook`. After the user resumes Emacs, `suspend-emacs` runs the normal hook `suspend-resume-hook`. See Section 24.1 [Hooks], page 525.

The next redisplay after resumption will redraw the entire screen, unless the variable `no-redraw-on-reenter` is non-`nil`. See Section 41.1 [Refresh Screen], page 1133.

Here is an example of how you could use these hooks:

```
(add-hook 'suspend-hook
          (lambda () (or (y-or-n-p "Really suspend?")
                         (error "Suspend canceled"))))
(add-hook 'suspend-resume-hook (lambda () (message "Resumed!")
                                  (sit-for 2)))
```

Here is what you would see upon evaluating `(suspend-emacs "pwd")`:

```
---------- Buffer: Minibuffer ----------
Really suspend? y
---------- Buffer: Minibuffer ----------

---------- Parent Shell ----------
bash$ /home/username
bash$ fg

---------- Echo Area ----------
Resumed!
```

Note that '`pwd`' is not echoed after Emacs is suspended. But it is read and executed by the shell.

`suspend-hook` [Variable]

This variable is a normal hook that Emacs runs before suspending.

`suspend-resume-hook` [Variable]

This variable is a normal hook that Emacs runs on resuming after a suspension.

`suspend-tty` **&optional** *tty* [Function]

If *tty* specifies a terminal device used by Emacs, this function relinquishes the device and restores it to its prior state. Frames that used the device continue to exist, but are not updated and Emacs doesn't read input from them. *tty* can be a terminal object, a frame (meaning the terminal for that frame), or `nil` (meaning the terminal for the selected frame). See Section 30.2 [Multiple Terminals], page 789.

If *tty* is already suspended, this function does nothing.

This function runs the hook `suspend-tty-functions`, passing the terminal object as an argument to each function.

`resume-tty` **&optional** *tty* [Function]
This function resumes the previously suspended terminal device *tty*; where *tty* has the same possible values as it does for `suspend-tty`.

This function reopens the terminal device, re-initializes it, and redraws it with that terminal's selected frame. It then runs the hook `resume-tty-functions`, passing the terminal object as an argument to each function.

If the same device is already used by another Emacs terminal, this function signals an error. If *tty* is not suspended, this function does nothing.

`controlling-tty-p` **&optional** *tty* [Function]
This function returns non-`nil` if *tty* is the controlling terminal of the Emacs session; *tty* can be a terminal object, a frame (meaning the terminal for that frame), or `nil` (meaning the terminal for the selected frame).

`suspend-frame` [Command]
This command *suspends* a frame. For GUI frames, it calls `iconify-frame` (see Section 30.11 [Visibility of Frames], page 831); for frames on text terminals, it calls either `suspend-emacs` or `suspend-tty`, depending on whether the frame is displayed on the controlling terminal device or not.

42.3 Operating System Environment

Emacs provides access to variables in the operating system environment through various functions. These variables include the name of the system, the user's UID, and so on.

`system-configuration` [Variable]
This variable holds the standard GNU configuration name for the hardware/software configuration of your system, as a string. For example, a typical value for a 64-bit GNU/Linux system is '`"x86_64-unknown-linux-gnu"`'.

`system-type` [Variable]
The value of this variable is a symbol indicating the type of operating system Emacs is running on. The possible values are:

`aix` IBM's AIX.

`berkeley-unix`
Berkeley BSD and its variants.

`cygwin` Cygwin, a POSIX layer on top of MS-Windows.

`darwin` Darwin (macOS).

`gnu` The GNU system (using the GNU kernel, which consists of the HURD and Mach).

`gnu/linux`
A GNU/Linux system—that is, a variant GNU system, using the Linux kernel. (These systems are the ones people often call "Linux", but actually Linux is just the kernel, not the whole system.)

`gnu/kfreebsd`
: A GNU (glibc-based) system with a FreeBSD kernel.

`haiku`
: The Haiku operating system, a derivative of the Be Operating System.

`hpux`
: Hewlett-Packard HPUX operating system.

`nacl`
: Google Native Client (NaCl) sandboxing system.

`ms-dos`
: Microsoft's DOS. Emacs compiled with DJGPP for MS-DOS binds `system-type` to `ms-dos` even when you run it on MS-Windows.

`usg-unix-v`
: AT&T Unix System V.

`windows-nt`
: Microsoft Windows NT, 9X and later. The value of `system-type` is always `windows-nt`, e.g., even on Windows 10.

We do not wish to add new symbols to make finer distinctions unless it is absolutely necessary! In fact, we hope to eliminate some of these alternatives in the future. If you need to make a finer distinction than `system-type` allows for, you can test `system-configuration`, e.g., against a regexp.

`system-name` [Function]
This function returns the name of the machine you are running on, as a string.

`mail-host-address` [User Option]
If this variable is non-`nil`, it is used instead of `system-name` for purposes of generating email addresses. For example, it is used when constructing the default value of `user-mail-address`. See Section 42.4 [User Identification], page 1274.

`getenv` *var* **&optional** *frame* [Command]
This function returns the value of the environment variable *var*, as a string. *var* should be a string. If *var* is undefined in the environment, `getenv` returns `nil`. It returns '`""`' if *var* is set but null. Within Emacs, a list of environment variables and their values is kept in the variable `process-environment`.

```
(getenv "USER")
     ⇒ "lewis"
```

The shell command `printenv` prints all or part of the environment:

```
bash$ printenv
PATH=/usr/local/bin:/usr/bin:/bin
USER=lewis
TERM=xterm
SHELL=/bin/bash
HOME=/home/lewis
...
```

`setenv` *variable* **&optional** *value substitute* [Command]
This command sets the value of the environment variable named *variable* to *value*. *variable* should be a string. Internally, Emacs Lisp can handle any string. However,

normally *variable* should be a valid shell identifier, that is, a sequence of letters, digits and underscores, starting with a letter or underscore. Otherwise, errors may occur if subprocesses of Emacs try to access the value of *variable*. If *value* is omitted or `nil` (or, interactively, with a prefix argument), `setenv` removes *variable* from the environment. Otherwise, *value* should be a string.

If the optional argument *substitute* is non-`nil`, Emacs calls the function `substitute-env-vars` to expand any environment variables in *value*.

`setenv` works by modifying `process-environment`; binding that variable with `let` is also reasonable practice.

`setenv` returns the new value of *variable*, or `nil` if it removed *variable* from the environment.

`with-environment-variables` *variables body...* [Macro]

This macro sets the environment variables according to *variables* temporarily when executing *body*. The previous values are restored when the form finishes. The argument *variables* should be a list of pairs of strings of the form `(var value)`, where *var* is the name of the environment variable and *value* is that variable's value.

```
(with-environment-variables (("LANG" "C")
                             ("LANGUAGE" "en_US:en"))
  (call-process "ls" nil t))
```

`process-environment` [Variable]

This variable is a list of strings, each describing one environment variable. The functions `getenv` and `setenv` work by means of this variable.

```
process-environment
⇒ ("PATH=/usr/local/bin:/usr/bin:/bin"
    "USER=lewis"
    "TERM=xterm"
    "SHELL=/bin/bash"
    "HOME=/home/lewis"
    ...)
```

If `process-environment` contains multiple elements that specify the same environment variable, the first of these elements specifies the variable, and the others are ignored.

`initial-environment` [Variable]

This variable holds the list of environment variables Emacs inherited from its parent process when Emacs started.

`path-separator` [Variable]

This variable holds a string that says which character separates directories in a search path (as found in an environment variable). Its value is `":"` for Unix and GNU systems, and `";"` for MS systems.

`path-separator` [Function]

This function returns the connection-local value of variable `path-separator`. That is `";"` for MS systems and a local `default-directory`, and `":"` for Unix and GNU systems, or a remote `default-directory`.

`parse-colon-path` *path* [Function]

This function takes a search path string such as the value of the `PATH` environment variable, and splits it at the separators, returning a list of directories. `nil` in this list means the current directory. Although the function's name says "colon", it actually uses the value of variable `path-separator`.

```
(parse-colon-path ":/foo:/bar")
     ⇒ (nil "/foo/" "/bar/")
```

`invocation-name` [Variable]

This variable holds the program name under which Emacs was invoked. The value is a string, and does not include a directory name.

`invocation-directory` [Variable]

This variable holds the directory in which the Emacs executable was located when it was run, or `nil` if that directory cannot be determined.

`installation-directory` [Variable]

If non-`nil`, this is a directory within which to look for the `lib-src` and `etc` subdirectories. In an installed Emacs, it is normally `nil`. It is non-`nil` when Emacs can't find those directories in their standard installed locations, but can find them in a directory related somehow to the one containing the Emacs executable (i.e., `invocation-directory`).

`load-average` **&optional** *use-float* [Function]

This function returns the current 1-minute, 5-minute, and 15-minute system load averages, in a list. The load average indicates the number of processes trying to run on the system.

By default, the values are integers that are 100 times the system load averages, but if *use-float* is non-`nil`, then they are returned as floating-point numbers without multiplying by 100.

If it is impossible to obtain the load average, this function signals an error. On some platforms, access to load averages requires installing Emacs as setuid or setgid so that it can read kernel information, and that usually isn't advisable.

If the 1-minute load average is available, but the 5- or 15-minute averages are not, this function returns a shortened list containing the available averages.

```
(load-average)
     ⇒ (169 48 36)
(load-average t)
     ⇒ (1.69 0.48 0.36)
```

The shell command `uptime` returns similar information.

`emacs-pid` [Function]

This function returns the process ID of the Emacs process, as an integer.

`tty-erase-char` [Variable]

This variable holds the erase character that was selected in the system's terminal driver, before Emacs was started.

`null-device` [Variable]
: This variable holds the system null device. Its value is `"/dev/null"` for Unix and GNU systems, and `"NUL"` for MS systems.

`null-device` [Function]
: This function returns the connection-local value of variable `null-device`. That is `"NUL"` for MS systems and a local `default-directory`, and `"/dev/null"` for Unix and GNU systems, or a remote `default-directory`.

42.4 User Identification

`init-file-user` [Variable]
: This variable says which user's init files should be used by Emacs—or `nil` if none. `""` stands for the user who originally logged in. The value reflects command-line options such as '`-q`' or '`-u` *user*'.

  Lisp packages that load files of customizations, or any other sort of user profile, should obey this variable in deciding where to find it. They should load the profile of the user name found in this variable. If `init-file-user` is `nil`, meaning that the '`-q`', '`-Q`', or '`-batch`' option was used, then Lisp packages should not load any customization files or user profile.

`user-mail-address` [User Option]
: This holds the email address of the user who is using Emacs.

`user-login-name` **&optional** *uid* [Function]
: This function returns the name under which the user is logged in. It uses the environment variables `LOGNAME` or `USER` if either is set. Otherwise, the value is based on the effective UID, not the real UID.

  If you specify *uid* (a number), the result is the user name that corresponds to *uid*, or `nil` if there is no such user.

`user-real-login-name` [Function]
: This function returns the user name corresponding to Emacs's real UID. This ignores the effective UID, and the environment variables `LOGNAME` and `USER`.

`user-full-name` **&optional** *uid* [Function]
: This function returns the full name of the logged-in user—or the value of the environment variable `NAME`, if that is set.

  If the Emacs process's user-id does not correspond to any known user (and provided `NAME` is not set), the result is `"unknown"`.

  If *uid* is non-`nil`, then it should be a number (a user-id) or a string (a login name). Then `user-full-name` returns the full name corresponding to that user-id or login name. If you specify a user-id or login name that isn't defined, it returns `nil`.

The symbols `user-login-name`, `user-real-login-name` and `user-full-name` are variables as well as functions. The functions return the same values that the variables hold. These variables allow you to fake out Emacs by telling the functions what to return. The variables are also useful for constructing frame titles (see Section 30.6 [Frame Titles], page 823).

`user-real-uid` [Function]
: This function returns the real UID of the user.

`user-uid` [Function]
: This function returns the effective UID of the user.

`group-gid` [Function]
: This function returns the effective GID of the Emacs process.

`group-real-gid` [Function]
: This function returns the real GID of the Emacs process.

`system-users` [Function]
: This function returns a list of strings, listing the user names on the system. If Emacs cannot retrieve this information, the return value is a list containing just the value of `user-real-login-name`.

`system-groups` [Function]
: This function returns a list of strings, listing the names of user groups on the system. If Emacs cannot retrieve this information, the return value is `nil`.

`group-name` *gid* [Function]
: This function returns the group name that corresponds to the numeric group ID *gid*, or `nil` if there is no such group.

42.5 Time of Day

This section explains how to determine the current time and time zone.

Many functions like `current-time` and `file-attributes` return *Lisp timestamp* values that count seconds, and that can represent absolute time by counting seconds since the *epoch* of 1970-01-01 00:00:00 UTC (Coordinated Universal Time). Typically these counts ignore leap seconds; however, GNU and some other operating systems can be configured to count leap seconds.

Although traditionally Lisp timestamps were integer pairs, their form has evolved and programs ordinarily should not depend on the current default form. If your program needs a particular timestamp form, you can use the `time-convert` function to convert it to the needed form. See Section 42.7 [Time Conversion], page 1278.

There are currently three forms of Lisp timestamps, each of which represents a number of seconds:

- An integer. Although this is the simplest form, it cannot represent subsecond timestamps.
- A pair of integers `(`*`ticks . hz`*`)`, where *hz* is positive. This represents *ticks*/*hz* seconds, which is the same time as plain *ticks* if *hz* is 1. A common value for *hz* is 1000000000, for a nanosecond-resolution clock.
- A list of four integers `(`*`high low micro pico`*`)`, where $0 \le low < 65536$, $0 \le micro < 1000000$, and $0 \le pico < 1000000$. This represents the number of seconds using the formula: $high \times 2^{16} + low + micro \times 10^{-6} + pico \times 10^{-12}$. If `current-time-list` is `t`, some functions may default to returning two- or three-element lists, with omitted *micro* and *pico*

components defaulting to zero. On all current machines *pico* is a multiple of 1000, but this may change as higher-resolution clocks become available.

Function arguments, e.g., the *time* argument to `format-time-string`, accept a more-general *time value* format, which can be a Lisp timestamp, `nil` for the current time, a finite floating-point number for seconds, or a list `(`*high low micro*`)` or `(`*high low*`)` that is a truncated list timestamp with missing elements taken to be zero.

Time values can be converted to and from calendrical and other forms. Some of these conversions rely on operating system functions that limit the range of possible time values, and signal an error such as '`"Specified time is not representable"`' if the limits are exceeded. For instance, a system might not support timestamps before the epoch, or years far in the future. You can convert a time value into a human-readable string using `format-time-string`, into a Lisp timestamp using `time-convert`, and into other forms using `decode-time` and `float-time`. These functions are described in the following sections.

`current-time-string` **&optional** *time zone* [Function]

This function returns the current time and date as a human-readable string. The format does not vary for the initial part of the string, which contains the day of week, month, day of month, and time of day in that order: the number of characters used for these fields is always the same, although (unless you require English weekday or month abbreviations regardless of locale) it is typically more convenient to use `format-time-string` than to extract fields from the output of `current-time-string`, as the year might not have exactly four digits, and additional information may some day be added at the end.

The argument *time*, if given, specifies a time to format, instead of the current time. The optional argument *zone* defaults to the current time zone rule. See Section 42.6 [Time Zone Rules], page 1277. The operating system limits the range of time and zone values.

```
(current-time-string)
     ⇒ "Fri Nov  1 15:59:49 2019"
```

`current-time-list` [Variable]

This boolean variable is a transition aid. If `t`, `current-time` and related functions return timestamps in list form, typically `(`*high low micro pico*`)`; otherwise, they use `(`*ticks* `.` *hz*`)` form. Currently this variable defaults to `t`, for behavior compatible with previous Emacs versions. Developers are encouraged to test timestamp-related code with this variable set to `nil`, as it will default to `nil` in a future Emacs version, and will be removed in some version after that.

`current-time` [Function]

This function returns the current time as a Lisp timestamp. If `current-time-list` is `nil`, the timestamp has the form `(`*ticks* `.` *hz*`)` where *ticks* counts clock ticks and *hz* is the clock ticks per second. Otherwise, the timestamp has the list form `(`*high low usec psec*`)`. You can use `(time-convert nil t)` or `(time-convert nil 'list)` to obtain a particular form regardless of the value of `current-time-list`. See Section 42.7 [Time Conversion], page 1278.

`float-time` **&optional** *time* [Function]

This function returns the current time as a floating-point number of seconds since the epoch. The optional argument *time*, if given, specifies a time to convert instead of the current time.

Warning: Since the result is floating point, it may not be exact. Do not use this function if precise time stamps are required. For example, on typical systems `(float-time '(1 . 10))` displays as '`0.1`' but is slightly greater than 1/10.

`time-to-seconds` is an alias for this function.

`current-cpu-time` [Function]

Return the current CPU time along with its resolution. The return value is a pair `(CPU-TICKS . TICKS-PER-SEC)`. The *CPU-TICKS* counter can wrap around, so values cannot be meaningfully compared if too much time has passed between them.

42.6 Time Zone Rules

The default time zone is determined by the `TZ` environment variable. See Section 42.3 [System Environment], page 1270. For example, you can tell Emacs to default to Universal Time with `(setenv "TZ" "UTC0")`. If `TZ` is not in the environment, Emacs uses system wall clock time, which is a platform-dependent default time zone.

The set of supported `TZ` strings is system-dependent. GNU and many other systems support TZDB timezones, e.g., '`"America/New_York"`' specifies the time zone and daylight saving time history for locations near New York City. GNU and most other systems support POSIX-style `TZ` strings, e.g., '`"EST5EDT,M4.1.0,M10.5.0"`' specifies the rules used in New York from 1987 through 2006. All systems support the string '`"UTC0"`' meaning Universal Time.

Functions that convert to and from local time accept an optional *time zone rule* argument, which specifies the conversion's time zone and daylight saving time history. If the time zone rule is omitted or `nil`, the conversion uses Emacs's default time zone. If it is `t`, the conversion uses Universal Time. If it is `wall`, the conversion uses the system wall clock time. If it is a string, the conversion uses the time zone rule equivalent to setting `TZ` to that string. If it is a list (*offset abbr*), where *offset* is an integer number of seconds east of Universal Time and *abbr* is a string, the conversion uses a fixed time zone with the given offset and abbreviation. An integer *offset* is treated as if it were (*offset abbr*), where *abbr* is a numeric abbreviation on POSIX-compatible platforms and is unspecified on MS-Windows.

`current-time-zone` **&optional** *time zone* [Function]

This function returns a list describing the time zone that the user is in.

The value has the form `(`*`offset abbr`*`)`. Here *offset* is an integer giving the number of seconds ahead of Universal Time (east of Greenwich). A negative value means west of Greenwich. The second element, *abbr*, is a string giving an abbreviation for the time zone, e.g., '`"CST"`' for China Standard Time or for U.S. Central Standard Time. Both elements can change when daylight saving time begins or ends; if the user has specified a time zone that does not use a seasonal time adjustment, then the value is constant through time.

If the operating system doesn't supply all the information necessary to compute the value, the unknown elements of the list are `nil`.

The argument *time*, if given, specifies a time value to analyze instead of the current time. The optional argument *zone* defaults to the current time zone rule. The operating system limits the range of time and zone values.

42.7 Time Conversion

These functions convert time values (see Section 42.5 [Time of Day], page 1275) to Lisp timestamps, or into calendrical information and vice versa.

Many operating systems use 64-bit signed integers to count seconds, and can represent times far in the past or future. However, some are more limited. For example, old-fashioned operating systems that use 32-bit signed integers typically handle only times from 1901-12-13 20:45:52 through 2038-01-19 03:14:07 Universal Time.

Calendrical conversion functions use the Gregorian calendar even for dates before the Gregorian calendar was introduced, and for dates in the far distant past or future for which the Gregorian calendar is wildly inaccurate and disagrees with common practice in scientific fields like astronomy and paleontology, which use Julian-calendar year lengths. Year numbers count since the year 1 BCE, and do not skip zero as traditional Gregorian years do; for example, the year number −37 represents the Gregorian year 38 BCE.

`time-convert` *time form* [Function]

This function converts a time value into a Lisp timestamp.

The *form* argument specifies the timestamp form to be returned. If *form* is the symbol `integer`, this function returns an integer count of seconds. If *form* is a positive integer, it specifies a clock frequency and this function returns an integer-pair timestamp (*ticks* . *form*). If *form* is `t`, this function treats it as a positive integer suitable for representing the timestamp; for example, it is treated as 1000000000 if *time* is `nil` and the platform timestamp has nanosecond resolution. If *form* is `list`, this function returns an integer list (*high low micro pico*). Although a `nil` *form* currently acts like `list`, this is planned to change in a future Emacs version, so callers requiring list timestamps should pass `list` explicitly.

If *time* is not a time value, this function signals an error. Otherwise, if *time* cannot be represented exactly, conversion truncates it toward minus infinity. When *form* is `t`, conversion is always exact so no truncation occurs, and the returned clock resolution is no less than that of *time*. By way of contrast, although `float-time` can also convert any time value without signaling an error, the result might not be exact. See Section 42.5 [Time of Day], page 1275.

For efficiency this function might return a value that is `eq` to *time*, or that otherwise shares structure with *time*.

Although `(time-convert nil nil)` is equivalent to `(current-time)`, the latter may be a bit faster.

```
(setq a (time-convert nil t))
⇒ (1564826753904873156 . 1000000000)
(time-convert a 100000)
⇒ (156482675390487 . 100000)
```

```
(time-convert a 'integer)
⇒ 1564826753
(time-convert a 'list)
⇒ (23877 23681 904873 156000)
```

`decode-time` **&optional** *time zone form* [Function]

This function converts a time value into calendrical information. If you don't specify *time*, it decodes the current time, and similarly *zone* defaults to the current time zone rule. See Section 42.6 [Time Zone Rules], page 1277. The operating system limits the range of time and zone values.

The *form* argument controls the form of the returned *seconds* element, as described below. The return value is a list of nine elements, as follows:

```
(seconds minutes hour day month year dow dst utcoff)
```

Here is what the elements mean:

seconds The number of seconds past the minute, with form described below.

minutes The number of minutes past the hour, as an integer between 0 and 59.

hour The hour of the day, as an integer between 0 and 23.

day The day of the month, as an integer between 1 and 31.

month The month of the year, as an integer between 1 and 12.

year The year, an integer typically greater than 1900.

dow The day of week, as an integer between 0 and 6, where 0 stands for Sunday.

dst `t` if daylight saving time is effect, `nil` if it is not in effect, and −1 if this information is not available.

utcoff An integer indicating the Universal Time offset in seconds, i.e., the number of seconds east of Greenwich.

The *seconds* element is a Lisp timestamp that is nonnegative and less than 61; it is less than 60 except during positive leap seconds (assuming the operating system supports leap seconds). If the optional *form* argument is `t`, *seconds* uses the same precision as *time*; if *form* is `integer`, *seconds* is truncated to an integer. For example, if *time* is the timestamp `(1566009571321 . 1000)`, which represents 2019-08-17 02:39:31.321 UTC on typical systems that lack leap seconds, then `(decode-time` *time* `t t)` returns `((31321 . 1000) 39 2 17 8 2019 6 nil 0)`, whereas `(decode-time` *time* `t 'integer)` returns `(31 39 2 17 8 2019 6 nil 0)`. If *form* is omitted or `nil`, it currently defaults to `integer` but this default may change in future Emacs releases, so callers requiring a particular form should specify *form*.

Common Lisp Note: Common Lisp has different meanings for *dow*, `dst` and *utcoff*, and its *second* is an integer between 0 and 59 inclusive.

To access (or alter) the elements in the calendrical information, the `decoded-time-second`, `decoded-time-minute`, `decoded-time-hour`, `decoded-time-day`, `decoded-time-month`, `decoded-time-year`, `decoded-time-weekday`, `decoded-time-dst` and `decoded-time-zone` accessors can be used.

`encode-time` *time* **&rest** *obsolescent-arguments* [Function]

This function converts *time* to a Lisp timestamp. It can act as the inverse of `decode-time`.

Ordinarily the first argument is a list `(second minute hour day month year ignored dst zone)` that specifies a decoded time in the style of `decode-time`. For the meanings of these list elements, see the table under `decode-time`. In particular, *dst* says how to interpret timestamps during a daylight saving fallback when timestamps are repeated. If *dst* is −1, the DST value is guessed; if it is `t` or `nil` the timestamp with that DST value is returned, with an error signaled if no such timestamp exists. Unfortunately a *dst* value of `t` or `nil` does not disambiguate timestamps duplicated when a TZDB-based timezone moves further west of Greenwich, such as disambiguating the two standard-time timestamps 2020-12-27 01:30 when *zone* is '`"Europe/Volgograd"`', which at 02:00 that day changed standard time from 4 to 3 hours east of Greenwich; if you need to handle situations like this you can use a numeric *zone* to disambiguate instead.

The first argument can also be a list `(second minute hour day month year)`, which is treated like the list `(second minute hour day month year nil -1 nil)`.

As an obsolescent calling convention, this function can be given six or more arguments. The first six arguments *second*, *minute*, *hour*, *day*, *month*, and *year* specify most of the components of a decoded time. If there are more than six arguments the *last* argument is used as *zone* and any other extra arguments are ignored, so that `(apply #'encode-time (decode-time ...))` works. In this obsolescent convention, *dst* is −1 and *zone* defaults to the current time zone rule (see Section 42.6 [Time Zone Rules], page 1277). When modernizing an obsolescent caller, ensure that the more-modern list equivalent contains 9 elements with a `dst` element that is −1, not `nil`.

Year numbers less than 100 are not treated specially. If you want them to stand for years above 1900, or years above 2000, you must alter them yourself before you call `encode-time`. The operating system limits the range of time and zone values. However, timestamps ranging from the epoch to the near future are always supported.

The `encode-time` function acts as a rough inverse to `decode-time`. For example, you can pass the output of the latter to the former as follows:

```
(encode-time (decode-time ...))
```

You can perform simple date arithmetic by using out-of-range values for *seconds*, *minutes*, *hour*, *day*, and *month*; for example, day 0 means the day preceding the given month. Take care when doing so, as it is common for this to fail in some cases. For example:

```
;; Try to compute the time one month from now.
;; Watch out; this might not work as expected.
(let ((time (decode-time)))
  (setf (decoded-time-month time)
        (+ (decoded-time-month time) 1))
  time)
```

Unfortunately, this code might not work as expected if the resulting time is invalid due to month length differences, daylight saving transitions, time zone changes, or missing leap days or leap seconds. For example, if executed on January 30 this code yields

a nonexistent date February 30, which `encode-time` would adjust to early March. Similarly, adding four years to February 29, 2096 would yield the nonexistent date February 29, 2100; and adding one hour to 01:30 on March 13, 2022 in New York would yield a timestamp 02:30 that does not exist because clocks sprang forward from 02:00 to 03:00 that day. To avoid some (though not all) of the problem, you can base calculations on the middle of the affected unit, e.g., start at the 15th of the month when adding months. Alternatively, you can use the `calendar` and `time-date` libraries.

42.8 Parsing and Formatting Times

These functions convert time values to text in a string, and vice versa. Time values are either represented as a Lisp timestamp (see Section 42.5 [Time of Day], page 1275) or a decoded time structure (see Section 42.7 [Time Conversion], page 1278).

`date-to-time` *string* [Function]

This function parses the time-string *string* and returns the corresponding Lisp timestamp. The argument *string* should represent a date-time, and should be in one of the forms recognized by `parse-time-string` (see below). This function assumes Universal Time if *string* lacks explicit time zone information, and assumes earliest values if *string* lacks month, day, or time. The operating system limits the range of time and zone values.

`parse-time-string` *string* [Function]

This function parses the time-string *string* into a decoded time structure (see Section 42.7 [Time Conversion], page 1278). The argument *string* should resemble an RFC 822 (or later) or ISO 8601 string, like "Fri, 25 Mar 2016 16:24:56 +0100" or "1998-09-12T12:21:54-0200", but this function will attempt to parse less well-formed time strings as well.

`iso8601-parse` *string* [Function]

For a more strict function (that will error out upon invalid input), this function can be used instead. It can parse all variants of the ISO 8601 standard, so in addition to the formats mentioned above, it also parses things like "1998W45-3" (week number) and "1998-245" (ordinal day number). To parse durations, there's `iso8601-parse-duration`, and to parse intervals, there's `iso8601-parse-interval`. All these functions return decoded time structures, except the final one, which returns three of them (the start, the end, and the duration).

`format-time-string` *format-string* **&optional** *time zone* [Function]

This function converts *time* (which should be a Lisp timestamp, and defaults to the current time if *time* is omitted or `nil`) to a string according to *format-string*. The conversion uses the time zone rule *zone*, which defaults to the current time zone rule. See Section 42.6 [Time Zone Rules], page 1277. The argument *format-string* may contain '`%`'-sequences which say to substitute parts of the time. Here is a table of what the '`%`'-sequences mean:

'`%a`' This stands for the abbreviated name of the day of week.

'`%A`' This stands for the full name of the day of week.

‘%b’ This stands for the abbreviated name of the month.

‘%B’ This stands for the full name of the month.

‘%c’ This is a synonym for ‘%x %X’.

‘%C’ This stands for the century, that is, the year divided by 100, truncated toward zero. The default field width is 2.

‘%d’ This stands for the day of month, zero-padded.

‘%D’ This is a synonym for ‘%m/%d/%y’.

‘%e’ This stands for the day of month, blank-padded.

‘%F’ This stands for the ISO 8601 date format, which is like ‘%+4Y-%m-%d’ except that any flags or field width override the ‘+’ and (after subtracting 6) the ‘4’.

‘%g’ This stands for the year without century (00–99) corresponding to the current *ISO week* number. ISO weeks start on Monday and end on Sunday. If an ISO week begins in one year and ends in another, the rules regarding which year ‘%g’ will produce are complex and will not be described here; however, in general, if most of the week’s days are in the ending year, ‘%g’ will produce that year.

‘%G’ This stands for the year with century corresponding to the current ISO week number.

‘%h’ This is a synonym for ‘%b’.

‘%H’ This stands for the hour (00–23).

‘%I’ This stands for the hour (01–12).

‘%j’ This stands for the day of the year (001–366).

‘%k’ This stands for the hour (0–23), blank padded.

‘%l’ This stands for the hour (1–12), blank padded.

‘%m’ This stands for the month (01–12).

‘%M’ This stands for the minute (00–59).

‘%n’ This stands for a newline.

‘%N’ This stands for the nanoseconds (000000000–999999999). To ask for fewer digits, use ‘%3N’ for milliseconds, ‘%6N’ for microseconds, etc. Any excess digits are discarded, without rounding.

‘%p’ This stands for ‘AM’ or ‘PM’, as appropriate.

‘%q’ This stands for the calendar quarter (1–4).

‘%r’ This is a synonym for ‘%I:%M:%S %p’.

‘%R’ This is a synonym for ‘%H:%M’.

‘%s’ This stands for the integer number of seconds since the epoch.

‘%S’ This stands for the second (00–59, or 00–60 on platforms that support leap seconds).

‘%t’ This stands for a tab character.

‘%T’ This is a synonym for ‘%H:%M:%S’.

‘%u’ This stands for the numeric day of week (1–7). Monday is day 1.

‘%U’ This stands for the week of the year (01–52), assuming that weeks start on Sunday.

‘%V’ This stands for the week of the year according to ISO 8601.

‘%w’ This stands for the numeric day of week (0–6). Sunday is day 0.

‘%W’ This stands for the week of the year (01–52), assuming that weeks start on Monday.

‘%x’ This has a locale-specific meaning. In the default locale (named ‘C’), it is equivalent to ‘%D’.

‘%X’ This has a locale-specific meaning. In the default locale (named ‘C’), it is equivalent to ‘%T’.

‘%y’ This stands for the year without century (00–99).

‘%Y’ This stands for the year with century.

‘%Z’ This stands for the time zone abbreviation (e.g., ‘EST’).

‘%z’ This stands for the time zone numerical offset. The ‘z’ can be preceded by one, two, or three colons; if plain ‘%z’ stands for ‘-0500’, then ‘%:z’ stands for ‘-05:00’, ‘%::z’ stands for ‘-05:00:00’, and ‘%:::z’ is like ‘%::z’ except it suppresses trailing instances of ‘:00’ so it stands for ‘-05’ in the same example.

‘%%’ This stands for a single ‘%’.

One or more flag characters can appear immediately after the ‘%’. ‘0’ pads with zeros, ‘+’ pads with zeros and also puts ‘+’ before nonnegative year numbers with more than four digits, ‘_’ pads with blanks, ‘-’ suppresses padding, ‘^’ upper-cases letters, and ‘#’ reverses the case of letters.

You can also specify the field width and type of padding for any of these ‘%’-sequences. This works as in `printf`: you write the field width as digits in a ‘%’-sequence, after any flags. For example, ‘%S’ specifies the number of seconds since the minute; ‘%03S’ means to pad this with zeros to 3 positions, ‘%_3S’ to pad with spaces to 3 positions. Plain ‘%3S’ pads with zeros, because that is how ‘%S’ normally pads to two positions.

The characters ‘E’ and ‘O’ act as modifiers when used after any flags and field widths in a ‘%’-sequence. ‘E’ specifies using the current locale’s alternative version of the date and time. In a Japanese locale, for example, %Ex might yield a date format based on the Japanese Emperors’ reigns. ‘E’ is allowed in ‘%Ec’, ‘%EC’, ‘%Ex’, ‘%EX’, ‘%Ey’, and ‘%EY’.

‘O’ means to use the current locale’s alternative representation of numbers, instead of the ordinary decimal digits. This is allowed with most letters, all the ones that output numbers.

To help debug programs, unrecognized '%'-sequences stand for themselves and are output as-is. Programs should not rely on this behavior, as future versions of Emacs may recognize new '%'-sequences as extensions.

This function uses the C library function `strftime` (see Section "Formatting Calendar Time" in *The GNU C Library Reference Manual*) to do most of the work. In order to communicate with that function, it first converts *time* and *zone* to internal form; the operating system limits the range of time and zone values. This function also encodes *format-string* using the coding system specified by `locale-coding-system` (see Section 34.12 [Locales], page 995); after `strftime` returns the resulting string, this function decodes the string using that same coding system.

`format-seconds` *format-string seconds* [Function]

This function converts its argument *seconds* into a string of years, days, hours, etc., according to *format-string*. The argument *format-string* may contain '%'-sequences which control the conversion. Here is a table of what the '%'-sequences mean:

'`%y`'
'`%Y`' The integer number of 365-day years.

'`%d`'
'`%D`' The integer number of days.

'`%h`'
'`%H`' The integer number of hours.

'`%m`'
'`%M`' The integer number of minutes.

'`%s`'
'`%S`' The number of seconds. If the optional '`,`' parameter is used, it's a floating point number, and the number after the '`,`' specifies how many decimals to be used. '`%,2s`' means "use two decimals".

'`%z`' Non-printing control flag. When it is used, other specifiers must be given in the order of decreasing size, i.e., years before days, hours before minutes, etc. Nothing will be produced in the result string to the left of '`%z`' until the first non-zero conversion is encountered. For example, the default format used by `emacs-uptime` (see Section 42.9 [Processor Run Time], page 1285) `"%Y, %D, %H, %M, %z%S"` means that the number of seconds will always be produced, but years, days, hours, and minutes will only be shown if they are non-zero.

'`%x`' Non-printing control flag that works along the same lines as '`%z`', but instead suppresses printing of trailing zero-value time elements.

'`%%`' Produces a literal '%'.

Upper-case format sequences produce the units in addition to the numbers, lower-case formats produce only the numbers.

You can also specify the field width by following the '%' with a number; shorter numbers will be padded with blanks. An optional period before the width requests zero-padding instead. For example, `"%.3Y"` might produce `"004 years"`.

42.9 Processor Run time

Emacs provides several functions and primitives that return time, both elapsed and processor time, used by the Emacs process.

`emacs-uptime` **&optional** *format* [Command]
: This function returns a string representing the Emacs *uptime*—the elapsed wall-clock time this instance of Emacs is running. The string is formatted by `format-seconds` according to the optional argument *format*. For the available format descriptors, see Section 42.8 [Time Parsing], page 1281. If *format* is `nil` or omitted, it defaults to `"%Y, %D, %H, %M, %z%S"`.

When called interactively, it prints the uptime in the echo area.

`get-internal-run-time` [Function]
: This function returns the processor run time used by Emacs, as a Lisp timestamp (see Section 42.5 [Time of Day], page 1275).

Note that the time returned by this function excludes the time Emacs was not using the processor, and if the Emacs process has several threads, the returned value is the sum of the processor times used up by all Emacs threads.

If the system doesn't provide a way to determine the processor run time, `get-internal-run-time` returns the same time as `current-time`.

`emacs-init-time` [Command]
: This function returns the duration of the Emacs initialization (see Section 42.1.1 [Startup Summary], page 1260) in seconds, as a string. When called interactively, it prints the duration in the echo area.

42.10 Time Calculations

These functions perform calendrical computations using time values (see Section 42.5 [Time of Day], page 1275). As with any time value, a value of `nil` for any of their time-value arguments stands for the current system time, and a finite number stands for the number of seconds since the epoch.

`time-less-p` *t1 t2* [Function]
: This returns `t` if the time value *t1* is less than the time value *t2*.

`time-equal-p` *t1 t2* [Function]
: This returns `t` if the two time values *t1* and *t2* are equal. The result is `nil` if either argument is a NaN. For the purpose of comparison, a `nil` argument represents the current time with infinite resolution, so this function returns `nil` if one argument is `nil` and the other is not, and callers can therefore use `nil` to represent an unknown time value that does not equal any timestamp.

`time-subtract` *t1 t2* [Function]
: This returns the time difference $t1 - t2$ between two time values, as a Lisp timestamp. The result is exact and its clock resolution is no worse than the worse of its two arguments' resolutions. If you need the difference in units of elapsed seconds, you can convert it with `time-convert` or `float-time`. See Section 42.7 [Time Conversion], page 1278.

`time-add` *t1 t2* [Function]
: This returns the sum of two time values, using the same conversion rules as `time-subtract`. One argument should represent a time difference rather than a point in time, as a time value that is often just a single number of elapsed seconds. Here is how to add a number of seconds to a time value:

```
(time-add time seconds)
```

`time-to-days` *time-value* [Function]
: This function returns the number of days between the beginning of year 1 and *time-value*, assuming the default time zone. The operating system limits the range of time and zone values.

`days-to-time` *days* [Function]
: This is not quite the inverse of the `time-to-days` function, as it uses the Emacs epoch (instead of the year 1) for historical reasons. To get the inverse, subtract `(time-to-days 0)` from *days*, in which case `days-to-time` may return `nil` if *days* is negative.

`time-to-day-in-year` *time-value* [Function]
: This returns the day number within the year corresponding to *time-value*, assuming the default time zone. The operating system limits the range of time and zone values.

`date-leap-year-p` *year* [Function]
: This function returns `t` if *year* is a leap year.

`date-days-in-month` *year month* [Function]
: Return the number of days in *month* in *year*. For instance, February 2020 has 29 days.

`date-ordinal-to-time` *year ordinal* [Function]
: Return the date of *ordinal* in *year* as a decoded time structure. For instance, the 120th day in 2004 is April 29th.

42.11 Timers for Delayed Execution

You can set up a *timer* to call a function at a specified future time or after a certain length of idleness. A timer is a special object that stores the information about the next invocation times and the function to invoke.

`timerp` *object* [Function]
: This predicate function returns non-`nil` if `object` is a timer.

Emacs cannot run timers at any arbitrary point in a Lisp program; it can run them only when Emacs could accept output from a subprocess: namely, while waiting or inside certain primitive functions such as `sit-for` or `read-event` which *can* wait. Therefore, a timer's execution may be delayed if Emacs is busy. However, the time of execution is very precise if Emacs is idle.

Emacs binds `inhibit-quit` to `t` before calling the timer function, because quitting out of many timer functions can leave things in an inconsistent state. This is normally unproblematical because most timer functions don't do a lot of work. Indeed, for a timer to call a

function that takes substantial time to run is likely to be annoying. If a timer function needs to allow quitting, it should use `with-local-quit` (see Section 22.11 [Quitting], page 472). For example, if a timer function calls `accept-process-output` to receive output from an external process, that call should be wrapped inside `with-local-quit`, to ensure that *C-g* works if the external process hangs.

It is usually a bad idea for timer functions to alter buffer contents. When they do, they usually should call `undo-boundary` both before and after changing the buffer, to separate the timer's changes from user commands' changes and prevent a single undo entry from growing to be quite large.

Timer functions should also avoid calling functions that cause Emacs to wait, such as `sit-for` (see Section 22.10 [Waiting], page 471). This can lead to unpredictable effects, since other timers (or even the same timer) can run while waiting. If a timer function needs to perform an action after a certain time has elapsed, it can do this by scheduling a new timer.

If a timer function performs a remote file operation, it can be in conflict with an already running remote file operation of the same connection. Such conflicts are detected, and they result in a `remote-file-error` error (see Appendix F [Standard Errors], page 1395). This should be protected by wrapping the timer function body with

```
(ignore-error 'remote-file-error
  ...)
```

If a timer function calls functions that can change the match data, it should save and restore the match data. See Section 35.6.4 [Saving Match Data], page 1019.

`run-at-time` *time repeat function* **&rest** *args* [Command]

This sets up a timer that calls the function *function* with arguments *args* at time *time*. If *repeat* is a number (integer or floating point), the timer is scheduled to run again every *repeat* seconds after *time*. If *repeat* is `nil`, the timer runs only once.

time may specify an absolute or a relative time.

Absolute times may be specified using a string with a limited variety of formats, and are taken to be times *today*, even if already in the past. The recognized forms are '*xxxx*', '*x*`:`*xx*', or '*xx*`:`*xx*' (military time), and '*xx*`am`', '*xx*`AM`', '*xx*`pm`', '*xx*`PM`', '*xx*`:`*xx*`am`', '*xx*`:`*xx*`AM`', '*xx*`:`*xx*`pm`', or '*xx*`:`*xx*`PM`'. A period can be used instead of a colon to separate the hour and minute parts.

To specify a relative time as a string, use numbers followed by units. For example:

'`1 min`' denotes 1 minute from now.

'`1 min 5 sec`'
denotes 65 seconds from now.

'`1 min 2 sec 3 hour 4 day 5 week 6 fortnight 7 month 8 year`'
denotes exactly 103 months, 123 days, and 10862 seconds from now.

For relative time values, Emacs considers a month to be exactly thirty days, and a year to be exactly 365.25 days.

Not all convenient formats are strings. If *time* is a number (integer or floating point), that specifies a relative time measured in seconds. The result of `encode-time` can also be used to specify an absolute value for *time*.

In most cases, *repeat* has no effect on when *first* call takes place—*time* alone specifies that. There is one exception: if *time* is `t`, then the timer runs whenever the time is a multiple of *repeat* seconds after the epoch. This is useful for functions like `display-time`. For instance, the following will make *function* run at every "whole" minute (e.g., '`11:03:00`', '`11:04:00`', etc):

```
(run-at-time t 60 function)
```

If Emacs didn't get any CPU time when the timer would have run (for example if the system was busy running another process or if the computer was sleeping or in a suspended state), the timer will run as soon as Emacs resumes and is idle.

The function `run-at-time` returns a timer value that identifies the particular scheduled future action. You can use this value to call `cancel-timer` (see below).

`run-with-timer` *secs repeat function* **&rest** *args* [Command]

This is exactly the same as `run-at-time` (so see that definition for an explanation of the parameters; *secs* is passed as *time* to that function), but is meant to be used when the delay is specified in seconds.

A repeating timer nominally ought to run every *repeat* seconds, but remember that any invocation of a timer can be late. Lateness of one repetition has no effect on the scheduled time of the next repetition. For instance, if Emacs is busy computing for long enough to cover three scheduled repetitions of the timer, and then starts to wait, it will immediately call the timer function three times in immediate succession (presuming no other timers trigger before or between them). If you want a timer to run again no less than *n* seconds after the last invocation, don't use the *repeat* argument. Instead, the timer function should explicitly reschedule the timer.

`timer-max-repeats` [User Option]

This variable's value specifies the maximum number of times to repeat calling a timer function in a row, when many previously scheduled calls were unavoidably delayed.

`with-timeout` (*seconds timeout-forms...*) *body...* [Macro]

Execute *body*, but give up after *seconds* seconds. If *body* finishes before the time is up, `with-timeout` returns the value of the last form in *body*. If, however, the execution of *body* is cut short by the timeout, then `with-timeout` executes all the *timeout-forms* and returns the value of the last of them.

This macro works by setting a timer to run after *seconds* seconds. If *body* finishes before that time, it cancels the timer. If the timer actually runs, it terminates execution of *body*, then executes *timeout-forms*.

Since timers can run within a Lisp program only when the program calls a primitive that can wait, `with-timeout` cannot stop executing *body* while it is in the midst of a computation—only when it calls one of those primitives. So use `with-timeout` only with a *body* that waits for input, not one that does a long computation.

The function `y-or-n-p-with-timeout` provides a simple way to use a timer to avoid waiting too long for an answer. See Section 21.7 [Yes-or-No Queries], page 414.

`cancel-timer` *timer* [Function]

This cancels the requested action for *timer*, which should be a timer—usually, one previously returned by `run-at-time` or `run-with-idle-timer`. This cancels the effect of that call to one of these functions; the arrival of the specified time will not cause anything special to happen.

The `list-timers` command lists all the currently active timers. The command `c` (`timer-list-cancel`) will cancel the timer on the line under point. You can sort the list by column using the command `S` (`tabulated-list-sort`).

42.12 Idle Timers

Here is how to set up a timer that runs when Emacs is idle for a certain length of time. Aside from how to set them up, idle timers work just like ordinary timers.

`run-with-idle-timer` *secs repeat function* **&rest** *args* [Command]

Set up a timer which runs the next time Emacs is idle for *secs* seconds. The value of *secs* may be a number or a value of the type returned by `current-idle-time`.

If *repeat* is `nil`, the timer runs just once, the first time Emacs remains idle for a long enough time. More often *repeat* is non-`nil`, which means to run the timer *each time* Emacs remains idle for *secs* seconds.

The function `run-with-idle-timer` returns a timer value which you can use in calling `cancel-timer` (see Section 42.11 [Timers], page 1286).

Emacs becomes *idle* when it starts waiting for user input (unless it waits for input with a timeout, see Section 22.8.2 [Reading One Event], page 463), and it remains idle until the user provides some input. If a timer is set for five seconds of idleness, it runs approximately five seconds after Emacs first becomes idle. Even if *repeat* is non-`nil`, this timer will not run again as long as Emacs remains idle, because the duration of idleness will continue to increase and will not go down to five seconds again.

Emacs can do various things while idle: garbage collect, autosave or handle data from a subprocess. But these interludes during idleness do not interfere with idle timers, because they do not reset the clock of idleness to zero. An idle timer set for 600 seconds will run when ten minutes have elapsed since the last user command was finished, even if subprocess output has been accepted thousands of times within those ten minutes, and even if there have been garbage collections and autosaves.

When the user supplies input, Emacs becomes non-idle while executing the input. Then it becomes idle again, and all the idle timers that are set up to repeat will subsequently run another time, one by one.

Do not write an idle timer function containing a loop which does a certain amount of processing each time around, and exits when `(input-pending-p)` is non-`nil`. This approach seems very natural but has two problems:

- It blocks out all process output (since Emacs accepts process output only while waiting).
- It blocks out any idle timers that ought to run during that time.

Similarly, do not write an idle timer function that sets up another idle timer (including the same idle timer) with *secs* argument less than or equal to the current idleness time.

Such a timer will run almost immediately, and continue running again and again, instead of waiting for the next time Emacs becomes idle. The correct approach is to reschedule with an appropriate increment of the current value of the idleness time, as described below.

`current-idle-time` [Function]

If Emacs is idle, this function returns the length of time Emacs has been idle, using the same format as `current-time` (see Section 42.5 [Time of Day], page 1275).

When Emacs is not idle, `current-idle-time` returns `nil`. This is a convenient way to test whether Emacs is idle.

The main use of `current-idle-time` is when an idle timer function wants to "take a break" for a while. It can set up another idle timer to call the same function again, after a few seconds more idleness. Here's an example:

```
(defvar my-resume-timer nil
  "Timer for `my-timer-function' to reschedule itself, or nil.")

(defun my-timer-function ()
  ;; If the user types a command while my-resume-timer
  ;; is active, the next time this function is called from
  ;; its main idle timer, deactivate my-resume-timer.
  (when my-resume-timer
    (cancel-timer my-resume-timer))
  ...do the work for a while...
  (when taking-a-break
    (setq my-resume-timer
          (run-with-idle-timer
            ;; Compute an idle time break-length
            ;; more than the current value.
            (time-add (current-idle-time) break-length)
            nil
            'my-timer-function))))
```

42.13 Terminal Input

This section describes functions and variables for recording or manipulating terminal input. See Chapter 41 [Display], page 1133, for related functions.

42.13.1 Input Modes

`set-input-mode` *interrupt flow meta* **&optional** *quit-char* [Function]

This function sets the mode for reading keyboard input. If *interrupt* is non-`nil`, then Emacs uses input interrupts. If it is `nil`, then it uses CBREAK mode. The default setting is system-dependent. Some systems always use CBREAK mode regardless of what is specified.

When Emacs communicates directly with X, it ignores this argument and uses interrupts if that is the way it knows how to communicate.

If *flow* is non-`nil`, then Emacs uses XON/XOFF (`C-q`, `C-s`) flow control for output to the terminal. This has no effect except in CBREAK mode.

The argument *meta* controls support for input character codes above 127. If *meta* is `t`, Emacs converts characters with the 8th bit set into Meta characters, before it decodes them as needed (see Section 34.10.8 [Terminal I/O Encoding], page 993). If *meta* is `nil`, Emacs disregards the 8th bit; this is necessary when the terminal uses it as a parity bit. If *meta* is the symbol `encoded`, Emacs first decodes the characters using all the 8 bits of each byte, and then converts the decoded single-byte characters into Meta characters if they have their eighth bit set. Finally, if *meta* is neither `t` nor `nil` nor `encoded`, Emacs uses all 8 bits of input unchanged, both before and after decoding them. This is good for terminals that use 8-bit character sets and don't encode the Meta modifier as the eighth bit.

If *quit-char* is non-`nil`, it specifies the character to use for quitting. Normally this character is `C-g`. See Section 22.11 [Quitting], page 472.

The `current-input-mode` function returns the input mode settings Emacs is currently using.

`current-input-mode` [Function]

This function returns the current mode for reading keyboard input. It returns a list, corresponding to the arguments of `set-input-mode`, of the form `(`*`interrupt flow meta quit`*`)` in which:

interrupt is non-`nil` when Emacs is using interrupt-driven input. If `nil`, Emacs is using CBREAK mode.

flow is non-`nil` if Emacs uses XON/XOFF (`C-q`, `C-s`) flow control for output to the terminal. This value is meaningful only when *interrupt* is `nil`.

meta is `t` if Emacs treats the eighth bit of input characters as the Meta bit before decoding input; `encoded` if Emacs treats the eighth bit of the decoded single-byte characters as the Meta bit; `nil` if Emacs clears the eighth bit of every input character; any other value means Emacs uses all eight bits as the basic character code.

quit is the character Emacs currently uses for quitting, usually `C-g`.

42.13.2 Recording Input

`recent-keys` **&optional** *include-cmds* [Function]

This function returns a vector containing the last 300 input events from the keyboard or mouse. All input events are included, whether or not they were used as parts of key sequences. Thus, you always get the last 300 input events, not counting events generated by keyboard macros. (These are excluded because they are less interesting for debugging; it should be enough to see the events that invoked the macros.)

If *include-cmds* is non-`nil`, complete key sequences in the result vector are interleaved with pseudo-events of the form `(nil . `*`COMMAND`*`)`, where *COMMAND* is the binding of the key sequence (see Section 22.1 [Command Overview], page 424).

A call to `clear-this-command-keys` (see Section 22.5 [Command Loop Info], page 437) causes this function to return an empty vector immediately afterward.

`open-dribble-file` *filename* [Command]
: This function opens a *dribble file* named *filename*. When a dribble file is open, each input event from the keyboard or mouse (but not those from keyboard macros) is written in that file. A non-character event is expressed using its printed representation surrounded by '`<...>`'. Be aware that sensitive information (such as passwords) may end up recorded in the dribble file.

  You close the dribble file by calling this function with an argument of `nil`.

See also the `open-termscript` function (see Section 42.14 [Terminal Output], page 1292).

42.14 Terminal Output

The terminal output functions send output to a text terminal, or keep track of output sent to the terminal. The variable `baud-rate` tells you what Emacs thinks is the output speed of the terminal.

`baud-rate` [User Option]
: This variable's value is the output speed of the terminal, as far as Emacs knows. Setting this variable does not change the speed of actual data transmission, but the value is used for calculations such as padding.

  It also affects decisions about whether to scroll part of the screen or repaint on text terminals. See Section 41.2 [Forcing Redisplay], page 1133, for the corresponding functionality on graphical terminals.

  The value is measured in baud.

If you are running across a network, and different parts of the network work at different baud rates, the value returned by Emacs may be different from the value used by your local terminal. Some network protocols communicate the local terminal speed to the remote machine, so that Emacs and other programs can get the proper value, but others do not. If Emacs has the wrong value, it makes decisions that are less than optimal. To fix the problem, set `baud-rate`.

`send-string-to-terminal` *string* **&optional** *terminal* [Function]
: This function sends *string* to *terminal* without alteration. Control characters in *string* have terminal-dependent effects. (If you need to display non-ASCII text on the terminal, encode it using one of the functions described in Section 34.10.7 [Explicit Encoding], page 991.) This function operates only on text terminals. *terminal* may be a terminal object, a frame, or `nil` for the selected frame's terminal. In batch mode, *string* is sent to `stdout` when *terminal* is `nil`.

  One use of this function is to define function keys on terminals that have downloadable function key definitions. For example, this is how (on certain terminals) to define function key 4 to move forward four characters (by transmitting the characters `C-u C-f` to the computer):

```
(send-string-to-terminal "\eF4\^U\^F")
     ⇒ nil
```

`open-termscript` *filename* [Command]
: This function is used to open a *termscript file* that will record all the characters sent by Emacs to the terminal. It returns `nil`. Termscript files are useful for investigating

problems where Emacs garbles the screen, problems that are due to incorrect Termcap entries or to undesirable settings of terminal options more often than to actual Emacs bugs. Once you are certain which characters were actually output, you can determine reliably whether they correspond to the Termcap specifications in use.

```
(open-termscript "../junk/termscript")
     ⇒ nil
```

You close the termscript file by calling this function with an argument of `nil`.

See also `open-dribble-file` in Section 42.13.2 [Recording Input], page 1291.

42.15 Sound Output

To play sound using Emacs, use the function `play-sound`. Only certain systems are supported; if you call `play-sound` on a system which cannot really do the job, it gives an error.

The sound must be stored as a file in RIFF-WAVE format (‘`.wav`’) or Sun Audio format (‘`.au`’).

`play-sound` *sound* [Function]

This function plays a specified sound. The argument, *sound*, has the form `(sound` *properties...*`)`, where the *properties* consist of alternating keywords (particular symbols recognized specially) and values corresponding to them.

Here is a table of the keywords that are currently meaningful in *sound*, and their meanings:

`:file` *file*
: This specifies the file containing the sound to play. If the file name is not absolute, it is expanded against the directory `data-directory`.

`:data` *data*
: This specifies the sound to play without need to refer to a file. The value, *data*, should be a string containing the same bytes as a sound file. We recommend using a unibyte string.

`:volume` *volume*
: This specifies how loud to play the sound. It should be a number in the range of 0 to 1. The default is to use whatever volume has been specified before.

`:device` *device*
: This specifies the system device on which to play the sound, as a string. The default device is system-dependent.

Before actually playing the sound, `play-sound` calls the functions in the list `play-sound-functions`. Each function is called with one argument, *sound*.

`play-sound-file` *file* **&optional** *volume device* [Command]

This function is an alternative interface to playing a sound *file* specifying an optional *volume* and *device*.

`play-sound-functions` [Variable]
: A list of functions to be called before playing a sound. Each function is called with one argument, a property list that describes the sound.

42.16 Operating on X11 Keysyms

To define system-specific X11 keysyms, set the variable `system-key-alist`.

`system-key-alist` [Variable]
: This variable's value should be an alist with one element for each system-specific keysym. Each element has the form `(code . symbol)`, where *code* is the numeric keysym code (not including the vendor-specific bit, -2^{28}), and *symbol* is the name for the function key.

  For example `(168 . mute-acute)` defines a system-specific key (used by HP X servers) whose numeric code is -2^{28} + 168.

  It is not crucial to exclude from the alist the keysyms of other X servers; those do no harm, as long as they don't conflict with the ones used by the X server actually in use.

  The variable is always local to the current terminal, and cannot be buffer-local. See Section 30.2 [Multiple Terminals], page 789.

You can specify which keysyms Emacs should use for the Control, Meta, Alt, Hyper, and Super modifiers by setting these variables:

`x-ctrl-keysym` [Variable]
`x-alt-keysym` [Variable]
`x-meta-keysym` [Variable]
`x-hyper-keysym` [Variable]
`x-super-keysym` [Variable]
: The name of the keysym that should stand for the Control modifier (respectively, for Alt, Meta, Hyper, and Super). For example, here is how to swap the Meta and Alt modifiers within Emacs:

```
(setq x-alt-keysym 'meta)
(setq x-meta-keysym 'alt)
```

42.17 Batch Mode

The command-line option '`-batch`' causes Emacs to run noninteractively. In this mode, Emacs does not read commands from the terminal, it does not alter the terminal modes, and it does not expect to be outputting to an erasable screen. The idea is that you specify Lisp programs to run; when they are finished, Emacs should exit. The way to specify the programs to run is with '`-l` *file*', which loads the library named *file*, or '`-f` *function*', which calls *function* with no arguments, or '`--eval=`*form*'.

`noninteractive` [Variable]
: This variable is non-`nil` when Emacs is running in batch mode.

If the specified Lisp program signals an unhandled error in batch mode, Emacs exits with a non-zero exit status after invoking the Lisp debugger which shows the Lisp backtrace (see Section 19.1.10 [Invoking the Debugger], page 341) on the standard error stream:

```
$ emacs -Q --batch --eval '(error "foo")'; echo $?

Error: error ("foo")
mapbacktrace(#f(compiled-function (evald func args flags) #<bytecode -0x4f85c5
7c45e2f81>))
debug-early-backtrace()
debug-early(error (error "foo"))
signal(error ("foo"))
error("foo")
eval((error "foo") t)
command-line-1(("--eval" "(error \"foo\")"))
command-line()
normal-top-level()
foo
255
```

Any Lisp program output that would normally go to the echo area, either using `message`, or using `prin1`, etc., with `t` as the stream (see Section 20.4 [Output Streams], page 376), goes instead to Emacs's standard descriptors when in batch mode: `message` writes to the standard error descriptor, while `prin1` and other print functions write to the standard output. Similarly, input that would normally come from the minibuffer is read from the standard input descriptor. Thus, Emacs behaves much like a noninteractive application program. (The echo area output that Emacs itself normally generates, such as command echoing, is suppressed entirely.)

Non-ASCII text written to the standard output or error descriptors is by default encoded using `locale-coding-system` (see Section 34.12 [Locales], page 995) if it is non-`nil`; this can be overridden by binding `coding-system-for-write` to a coding system of you choice (see Section 34.10.7 [Explicit Encoding], page 991).

In batch mode, Emacs will enlarge the value of the `gc-cons-percentage` variable from the default of '`0.1`' up to '`1.0`'. Batch jobs that are supposed to run for a long time should adjust the limit back down again, because this means that less garbage collection will be performed by default (and more memory consumed).

42.18 Session Management

Emacs supports the X Session Management Protocol, which is used to suspend and restart applications. In the X Window System, a program called the *session manager* is responsible for keeping track of the applications that are running. When the X server shuts down, the session manager asks applications to save their state, and delays the actual shutdown until they respond. An application can also cancel the shutdown.

When the session manager restarts a suspended session, it directs these applications to individually reload their saved state. It does this by specifying a special command-line argument that says what saved session to restore. For Emacs, this argument is '`--smid` *`session`*'.

`emacs-save-session-functions` [Variable]

Emacs supports saving state via a hook called `emacs-save-session-functions`. Emacs runs this hook when the session manager tells it that the window system is shutting down. The functions are called with no arguments, and with the current buffer set to a temporary buffer. Each function can use `insert` to add Lisp code to this buffer. At the end, Emacs saves the buffer in a file, called the *session file*.

Subsequently, when the session manager restarts Emacs, it loads the session file automatically (see Chapter 16 [Loading], page 299). This is performed by a function named `emacs-session-restore`, which is called during startup. See Section 42.1.1 [Startup Summary], page 1260.

If a function in `emacs-save-session-functions` returns non-`nil`, Emacs tells the session manager to cancel the shutdown.

Here is an example that just inserts some text into `*scratch*` when Emacs is restarted by the session manager.

```
(add-hook 'emacs-save-session-functions 'save-yourself-test)

(defun save-yourself-test ()
  (insert
   (format "%S" '(with-current-buffer "*scratch*"
                   (insert "I am restored"))))
  nil)
```

42.19 Desktop Notifications

Emacs is able to send *notifications* on systems that support the freedesktop.org Desktop Notifications Specification and on MS-Windows. In order to use this functionality on POSIX hosts, Emacs must have been compiled with D-Bus support, and the `notifications` library must be loaded. See Section "D-Bus" in *D-Bus integration in Emacs*. The following function is supported when D-Bus support is available:

`notifications-notify` **&rest** *params* [Function]

This function sends a notification to the desktop via D-Bus, consisting of the parameters specified by the *params* arguments. These arguments should consist of alternating keyword and value pairs. The supported keywords and values are as follows:

`:bus` *bus* The D-Bus bus. This argument is needed only if a bus other than `:session` shall be used.

`:title` *title*

The notification title.

`:body` *text*

The notification body text. Depending on the implementation of the notification server, the text could contain HTML markups, like '`"<b>bold text</b>"`', hyperlinks, or images. Special HTML characters must be encoded, as '`"Contact <postmaster@localhost>!"`'.

`:app-name` *name*
: The name of the application sending the notification. The default is `notifications-application-name`.

`:replaces-id` *id*
: The notification *id* that this notification replaces. *id* must be the result of a previous `notifications-notify` call.

`:app-icon` *icon-file*
: The file name of the notification icon. If set to `nil`, no icon is displayed. The default is `notifications-application-icon`.

`:actions (`*key title key title* `...)`
: A list of actions to be applied. *key* and *title* are both strings. The default action (usually invoked by clicking the notification) should have a key named '`"default"`'. The title can be anything, though implementations are free not to display it.

`:timeout` *timeout*
: The timeout time in milliseconds since the display of the notification at which the notification should automatically close. If -1, the notification's expiration time is dependent on the notification server's settings, and may vary for the type of notification. If 0, the notification never expires. Default value is -1.

`:urgency` *urgency*
: The urgency level. It can be `low`, `normal`, or `critical`.

`:action-items`
: When this keyword is given, the *title* string of the actions is interpreted as icon name.

`:category` *category*
: The type of notification this is, a string. See the Desktop Notifications Specification (`https://specifications.freedesktop.org/notification-spec/notification-spec-latest.html#categories`) for a list of standard categories.

`:desktop-entry` *filename*
: This specifies the name of the desktop filename representing the calling program, like '`"emacs"`'.

`:image-data (`*width height rowstride has-alpha bits channels data*`)`
: This is a raw data image format that describes the width, height, rowstride, whether there is an alpha channel, bits per sample, channels and image data, respectively.

`:image-path` *path*
: This is represented either as a URI ('`file://`' is the only URI schema supported right now) or a name in a freedesktop.org-compliant icon theme from '`$XDG_DATA_DIRS/icons`'.

`:sound-file` *filename*
: The path to a sound file to play when the notification pops up.

`:sound-name` *name*
: A themable named sound from the freedesktop.org sound naming specification from '`$XDG_DATA_DIRS/sounds`', to play when the notification pops up. Similar to the icon name, only for sounds. An example would be '`"message-new-instant"`'.

`:suppress-sound`
: Causes the server to suppress playing any sounds, if it has that ability.

`:resident`
: When set the server will not automatically remove the notification when an action has been invoked. The notification will remain resident in the server until it is explicitly removed by the user or by the sender. This hint is likely only useful when the server has the `:persistence` capability.

`:transient`
: When set the server will treat the notification as transient and by-pass the server's persistence capability, if it should exist.

`:x` *position*
`:y` *position*
: Specifies the X, Y location on the screen that the notification should point to. Both arguments must be used together.

`:on-action` *function*
: Function to call when an action is invoked. The notification *id* and the *key* of the action are passed as arguments to the function.

`:on-close` *function*
: Function to call when the notification has been closed by timeout or by the user. The function receive the notification *id* and the closing *reason* as arguments:
  - `expired` if the notification has expired
  - `dismissed` if the notification was dismissed by the user
  - `close-notification` if the notification was closed by a call to `notifications-close-notification`
  - `undefined` if the notification server hasn't provided a reason

Which parameters are accepted by the notification server can be checked via `notifications-get-capabilities`.

This function returns a notification id, an integer, which can be used to manipulate the notification item with `notifications-close-notification` or the `:replaces-id` argument of another `notifications-notify` call. For example:

```
(defun my-on-action-function (id key)
  (message "Message %d, key \"%s\" pressed" id key))
     ⇒ my-on-action-function

(defun my-on-close-function (id reason)
  (message "Message %d, closed due to \"%s\"" id reason))
     ⇒ my-on-close-function
```

```
(notifications-notify
 :title "Title"
 :body "This is <b>important</b>."
 :actions '("Confirm" "I agree" "Refuse" "I disagree")
 :on-action 'my-on-action-function
 :on-close 'my-on-close-function)
     ⇒ 22

A message window opens on the desktop.  Press ``I agree''.
     ⇒ Message 22, key "Confirm" pressed
       Message 22, closed due to "dismissed"
```

`notifications-close-notification` *id* **&optional** *bus* [Function]

This function closes a notification with identifier *id*. *bus* can be a string denoting a D-Bus connection, the default is `:session`.

`notifications-get-capabilities` **&optional** *bus* [Function]

Returns the capabilities of the notification server, a list of symbols. *bus* can be a string denoting a D-Bus connection, the default is `:session`. The following capabilities can be expected:

`:actions` The server will provide the specified actions to the user.

`:body` Supports body text.

`:body-hyperlinks`
The server supports hyperlinks in the notifications.

`:body-images`
The server supports images in the notifications.

`:body-markup`
Supports markup in the body text.

`:icon-multi`
The server will render an animation of all the frames in a given image array.

`:icon-static`
Supports display of exactly 1 frame of any given image array. This value is mutually exclusive with `:icon-multi`.

`:persistence`
The server supports persistence of notifications.

`:sound` The server supports sounds on notifications.

Further vendor-specific caps start with `:x-vendor`, like `:x-gnome-foo-cap`.

`notifications-get-server-information` **&optional** *bus* [Function]

Return information on the notification server, a list of strings. *bus* can be a string denoting a D-Bus connection, the default is `:session`. The returned list is (*name vendor version spec-version*).

name The product name of the server.

vendor The vendor name. For example, '`"KDE"`', '`"GNOME"`'.

version The server's version number.

spec-version
The specification version the server is compliant with.

If *spec_version* is `nil`, the server supports a specification prior to '`"1.0"`'.

When Emacs runs on MS-Windows as a GUI session, it supports a small subset of the D-Bus notifications functionality via a native primitive:

`w32-notification-notify` **&rest** *params* [Function]
This function displays an MS-Windows tray notification as specified by *params*. MS-Windows tray notifications are displayed in a balloon from an icon in the notification area of the taskbar.

Value is the integer unique ID of the notification that can be used to remove the notification using `w32-notification-close`, described below. If the function fails, the return value is `nil`.

The arguments *params* are specified as keyword/value pairs. All the parameters are optional, but if no parameters are specified, the function will do nothing and return `nil`.

The following parameters are supported:

`:icon` *icon*
Display *icon* in the system tray. If *icon* is a string, it should specify a file name from which to load the icon; the specified file should be a `.ico` Windows icon file. If *icon* is not a string, or if this parameter is not specified, the standard Emacs icon will be used.

`:tip` *tip* Use *tip* as the tooltip for the notification. If *tip* is a string, this is the text of a tooltip that will be shown when the mouse pointer hovers over the tray icon added by the notification. If *tip* is not a string, or if this parameter is not specified, the default tooltip text is '`Emacs notification`'. The tooltip text can be up to 127 characters long (63 on Windows versions before W2K). Longer strings will be truncated.

`:level` *level*
Notification severity level, one of `info`, `warning`, or `error`. If given, the value determines the icon displayed to the left of the notification title, but only if the `:title` parameter (see below) is also specified and is a string.

`:title` *title*
The title of the notification. If *title* is a string, it is displayed in a larger font immediately above the body text. The title text can be up to 63 characters long; longer text will be truncated.

`:body` *body*
The body of the notification. If *body* is a string, it specifies the text of the notification message. Use embedded newlines to control how the text

is broken into lines. The body text can be up to 255 characters long, and will be truncated if it's longer. Unlike with D-Bus, the body text should be plain text, with no markup.

Note that versions of Windows before W2K support only `:icon` and `:tip`. The other parameters can be passed, but they will be ignored on those old systems.

There can be at most one active notification at any given time. An active notification must be removed by calling `w32-notification-close` before a new one can be shown.

To remove the notification and its icon from the taskbar, use the following function:

`w32-notification-close` *id* [Function]
This function removes the tray notification given by its unique *id*.

42.20 Notifications on File Changes

Several operating systems support watching of filesystems for changes to files or their attributes. If configured properly, Emacs links a respective library like `inotify`, `kqueue`, `gfilenotify`, or `w32notify` statically. These libraries enable watching of filesystems on the local machine.

It is also possible to watch filesystems on remote machines, see Section "Remote Files" in *The GNU Emacs Manual*. This does not depend on one of the libraries linked to Emacs.

Since all these libraries emit different events upon notified file changes, Emacs provides a special library `filenotify` which presents a unified interface to applications. Lisp programs that want to receive file notifications should always use this library in preference to the native ones. This section documents the `filenotify` library functions and variables.

`file-notify-add-watch` *file flags callback* [Function]
Add a watch for filesystem events pertaining to *file*. This arranges for filesystem events pertaining to *file* to be reported to Emacs.

The returned value is a descriptor for the added watch. Its type depends on the underlying library, and in general cannot be assumed to be an integer as in the example below. It should be used for comparison by `equal` only.

If the *file* cannot be watched for some reason, this function signals a `file-notify-error` error.

Sometimes, mounted filesystems cannot be watched for file changes. This is not detected by this function, and so a non-`nil` return value does not guarantee that changes on *file* will be actually notified.

If *file* is a symlink, it doesn't follow that link. Just *file* itself will be watched.

flags is a list of conditions to set what will be watched for. It can include the following symbols:

`change` watch for changes in file's contents

`attribute-change`
watch for changes in file attributes, like permissions or modification time

If *file* is a directory, `change` watches for file creation and deletion in that directory. Some of the native file notification libraries also report file changes in that case. This does not work recursively.

When any event happens, Emacs will call the *callback* function passing it a single argument *event*, which is of the form

```
(descriptor action file [file1])
```

descriptor is the same object as the one returned by this function. *action* is the description of the event. It could be any one of the following symbols:

`created` *file* was created

`deleted` *file* was deleted

`changed` *file*'s contents has changed; with `w32notify` library, reports attribute changes as well

`renamed` *file* has been renamed to *file1*

`attribute-changed`
a *file* attribute was changed

`stopped` watching *file* has stopped

Note that the `w32notify` library does not report `attribute-changed` events. When some file's attribute, like permissions or modification time, has changed, this library reports a `changed` event. Likewise, the `kqueue` library does not reliably report file attribute changes when watching a directory.

The `stopped` event means that watching the file has been discontinued. This could be because `file-notify-rm-watch` was called (see below), or because the file being watched was deleted, or due to another error reported from the underlying library which makes further watching impossible.

file and *file1* are the name of the file(s) whose event is being reported. For example:

```
(require 'filenotify)
     ⇒ filenotify

(defun my-notify-callback (event)
  (message "Event %S" event))
     ⇒ my-notify-callback

(file-notify-add-watch
  "/tmp" '(change attribute-change) 'my-notify-callback)
     ⇒ 35025468

(write-region "foo" nil "/tmp/foo")
     ⇒ Event (35025468 created "/tmp/.#foo")
        Event (35025468 created "/tmp/foo")
        Event (35025468 changed "/tmp/foo")
        Event (35025468 deleted "/tmp/.#foo")
```

```
(write-region "bla" nil "/tmp/foo")
     ⇒ Event (35025468 created "/tmp/.#foo")
        Event (35025468 changed "/tmp/foo")
        Event (35025468 deleted "/tmp/.#foo")

(set-file-modes "/tmp/foo" (default-file-modes) 'nofollow)
     ⇒ Event (35025468 attribute-changed "/tmp/foo")
```

Whether the action `renamed` is returned depends on the used watch library. Otherwise, the actions `deleted` and `created` could be returned in a random order.

```
(rename-file "/tmp/foo" "/tmp/bla")
     ⇒ Event (35025468 renamed "/tmp/foo" "/tmp/bla")

(delete-file "/tmp/bla")
     ⇒ Event (35025468 deleted "/tmp/bla")
```

`file-notify-rm-watch` *descriptor* [Function]

Removes an existing file watch specified by its *descriptor*. *descriptor* should be an object returned by `file-notify-add-watch`.

`file-notify-rm-all-watches` [Command]

Removes all existing file notification watches from Emacs.

Use this command with caution, because it could have unexpected side effects on packages relying on file watches. It is intended mainly for debugging purposes, or when Emacs has been stalled.

`file-notify-valid-p` *descriptor* [Function]

Checks a watch specified by its *descriptor* for validity. *descriptor* should be an object returned by `file-notify-add-watch`.

A watch can become invalid if the file or directory it watches is deleted, or if the watcher thread exits abnormally for any other reason. Removing the watch by calling `file-notify-rm-watch` also makes it invalid.

```
(make-directory "/tmp/foo")
     ⇒ Event (35025468 created "/tmp/foo")

(setq desc
      (file-notify-add-watch
        "/tmp/foo" '(change) 'my-notify-callback))
     ⇒ 11359632

(file-notify-valid-p desc)
     ⇒ t

(write-region "bla" nil "/tmp/foo/bla")
     ⇒ Event (11359632 created "/tmp/foo/.#bla")
        Event (11359632 created "/tmp/foo/bla")
        Event (11359632 changed "/tmp/foo/bla")
        Event (11359632 deleted "/tmp/foo/.#bla")
```

```
;; Deleting a file in the directory doesn't invalidate the watch.
(delete-file "/tmp/foo/bla")
     ⇒ Event (11359632 deleted "/tmp/foo/bla")

(write-region "bla" nil "/tmp/foo/bla")
     ⇒ Event (11359632 created "/tmp/foo/.#bla")
        Event (11359632 created "/tmp/foo/bla")
        Event (11359632 changed "/tmp/foo/bla")
        Event (11359632 deleted "/tmp/foo/.#bla")

;; Deleting the directory invalidates the watch.
;; Events arrive for different watch descriptors.
(delete-directory "/tmp/foo" 'recursive)
     ⇒ Event (35025468 deleted "/tmp/foo")
        Event (11359632 deleted "/tmp/foo/bla")
        Event (11359632 deleted "/tmp/foo")
        Event (11359632 stopped "/tmp/foo")

(file-notify-valid-p desc)
     ⇒ nil
```

42.21 Dynamically Loaded Libraries

A *dynamically loaded library* is a library that is loaded on demand, when its facilities are first needed. Emacs supports such on-demand loading of support libraries for some of its features.

`dynamic-library-alist` [Variable]

This is an alist of dynamic libraries and external library files implementing them.

Each element is a list of the form (*`library files...`*), where the `car` is a symbol representing a supported external library, and the rest are strings giving alternate filenames for that library.

Emacs tries to load the library from the files in the order they appear in the list; if none is found, the Emacs session won't have access to that library, and the features it provides will be unavailable.

Image support on some platforms uses this facility. Here's an example of setting this variable for supporting images on MS-Windows:

```
(setq dynamic-library-alist
      '((xpm "libxpm.dll" "xpm4.dll" "libXpm-nox4.dll")
        (png "libpng12d.dll" "libpng12.dll" "libpng.dll"
             "libpng13d.dll" "libpng13.dll")
        (jpeg "jpeg62.dll" "libjpeg.dll" "jpeg-62.dll"
              "jpeg.dll")
        (tiff "libtiff3.dll" "libtiff.dll")
        (gif "giflib4.dll" "libungif4.dll" "libungif.dll")
        (svg "librsvg-2-2.dll")
```

```
(gdk-pixbuf "libgdk_pixbuf-2.0-0.dll")
(glib "libglib-2.0-0.dll")
(gobject "libgobject-2.0-0.dll")))
```

Note that image types `pbm` and `xbm` do not need entries in this variable because they do not depend on external libraries and are always available in Emacs.

Also note that this variable is not meant to be a generic facility for accessing external libraries; only those already known by Emacs can be loaded through it.

This variable is ignored if the given *library* is statically linked into Emacs.

42.22 Security Considerations

Like any application, Emacs can be run in a secure environment, where the operating system enforces rules about access and the like. With some care, Emacs-based applications can also be part of a security perimeter that checks such rules. Although the default settings for Emacs work well for a typical software development environment, they may require adjustment in environments containing untrusted users that may include attackers. Here is a compendium of security issues that may be helpful if you are developing such applications. It is by no means complete; it is intended to give you an idea of the security issues involved, rather than to be a security checklist.

File local variables
: A file that Emacs visits can contain variable settings that affect the buffer visiting that file; See Section 12.12 [File Local Variables], page 213. Similarly, a directory can specify local variable values common to all files in that directory; see Section 12.13 [Directory Local Variables], page 217. Although Emacs takes some effort to protect against misuse of these variables, a security hole can be created merely by a package setting `safe-local-variable` too optimistically, a problem that is all too common. To disable this feature for both files and directories, set `enable-local-variables` to `nil`.

Access control
: Although Emacs normally respects access permissions of the underlying operating system, in some cases it handles accesses specially. For example, file names can have handlers that treat the files specially, with their own access checking. See Section 26.12 [Magic File Names], page 651. Also, a buffer can be read-only even if the corresponding file is writable, and vice versa, which can result in messages such as '`File passwd is write-protected; try to save anyway? (yes or no)`'. See Section 28.7 [Read Only Buffers], page 682.

Authentication
: Emacs has several functions that deal with passwords, e.g., `read-passwd`. See Section 21.9 [Reading a Password], page 418. Although these functions do not attempt to broadcast passwords to the world, their implementations are not proof against determined attackers with access to Emacs internals. For example, even if Elisp code uses `clear-string` to scrub a password from its memory after using it, remnants of the password may still reside in the garbage-collected free list. See Section 4.4 [Modifying Strings], page 59.

Code injection
: Emacs can send commands to many other applications, and applications should take care that strings sent as operands of these commands are not misinterpreted as directives. For example, when using a shell command to rename a file *a* to *b*, do not simply use the string `mv a b`, because either file name might start with '`-`', or might contain shell metacharacters like '`;`'. Although functions like `shell-quote-argument` can help avoid this sort of problem, they are not panaceas; for example, on a POSIX platform `shell-quote-argument` quotes shell metacharacters but not leading '`-`'. On MS-Windows, quoting for '`%`' assumes none of the environment variables have '`^`' in their name. See Section 40.2 [Shell Arguments], page 1083. Typically it is safer to use `call-process` than a subshell. See Section 40.3 [Synchronous Processes], page 1084. And it is safer yet to use builtin Emacs functions; for example, use `(rename-file "a" "b" t)` instead of invoking `mv`. See Section 26.7 [Changing Files], page 631.

Coding systems
: Emacs attempts to infer the coding systems of the files and network connections it accesses. See Section 34.10 [Coding Systems], page 980. If Emacs infers incorrectly, or if the other parties to the network connection disagree with Emacs's inferences, the resulting system could be unreliable. Also, even when it infers correctly, Emacs often can use bytes that other programs cannot. For example, although to Emacs the null byte is just a character like any other, many other applications treat it as a string terminator and mishandle strings or files containing null bytes.

Environment and configuration variables
: POSIX specifies several environment variables that can affect how Emacs behaves. Any environment variable whose name consists entirely of uppercase ASCII letters, digits, and the underscore may affect the internal behavior of Emacs. Emacs uses several such variables, e.g., `EMACSLOADPATH`. See Section 16.3 [Library Search], page 302. On some platforms some environment variables (e.g., `PATH`, `POSIXLY_CORRECT`, `SHELL`, `TMPDIR`) need to have properly-configured values in order to get standard behavior for any utility Emacs might invoke. Even seemingly-benign variables like `TZ` may have security implications. See Section 42.3 [System Environment], page 1270.

  Emacs has customization and other variables with similar considerations. For example, if the variable `shell-file-name` specifies a shell with nonstandard behavior, an Emacs-based application may misbehave.

Installation
: When Emacs is installed, if the installation directory hierarchy can be modified by untrusted users, the application cannot be trusted. This applies also to the directory hierarchies of the programs that Emacs uses, and of the files that Emacs reads and writes.

Network access
: Emacs often accesses the network, and you may want to configure it to avoid network accesses that it would normally do. For example, unless you set

`tramp-mode` to `nil`, file names using a certain syntax are interpreted as being network files, and are retrieved across the network. See *The Tramp Manual*.

Race conditions

: Emacs applications have the same sort of race-condition issues that other applications do. For example, even when `(file-readable-p "foo.txt")` returns `t`, it could be that `foo.txt` is unreadable because some other program changed the file's permissions between the call to `file-readable-p` and now. See Section 26.6.1 [Testing Accessibility], page 621.

Resource limits

: When Emacs exhausts memory or other operating system resources, its behavior can be less reliable, in that computations that ordinarily run to completion may abort back to the top level. This may cause Emacs to neglect operations that it normally would have done.

43 Preparing Lisp code for distribution

Emacs provides a standard way to distribute Emacs Lisp code to users. A *package* is a collection of one or more files, formatted and bundled in such a way that users can easily download, install, uninstall, and upgrade it.

The following sections describe how to create a package, and how to put it in a *package archive* for others to download. See Section "Packages" in *The GNU Emacs Manual*, for a description of user-level features of the packaging system.

These sections are mostly directed towards package archive maintainers—much of this information is not relevant for package authors (i.e., people who write code that will be distributed via these archives).

43.1 Packaging Basics

A package is either a *simple package* or a *multi-file package*. A simple package is stored in a package archive as a single Emacs Lisp file, while a multi-file package is stored as a tar file (containing multiple Lisp files, and possibly non-Lisp files such as a manual).

In ordinary usage, the difference between simple packages and multi-file packages is relatively unimportant; the Package Menu interface makes no distinction between them. However, the procedure for creating them differs, as explained in the following sections.

Each package (whether simple or multi-file) has certain *attributes*:

Name
: A short word (e.g., '`auctex`'). This is usually also the symbol prefix used in the program (see Section D.1 [Coding Conventions], page 1336).

Version
: A version number, in a form that the function `version-to-list` understands (e.g., '`11.86`'). Each release of a package should be accompanied by an increase in the version number so that it will be recognized as an upgrade by users querying the package archive.

Brief description
: This is shown when the package is listed in the Package Menu. It should occupy a single line, ideally in 36 characters or less.

Long description
: This is shown in the buffer created by *C-h P* (`describe-package`), following the package's brief description and installation status. It normally spans multiple lines, and should fully describe the package's capabilities and how to begin using it once it is installed.

Dependencies
: A list of other packages (possibly including minimal acceptable version numbers) on which this package depends. The list may be empty, meaning this package has no dependencies. Otherwise, installing this package also automatically installs its dependencies, recursively; if any dependency cannot be found, the package cannot be installed.

Installing a package, either via the command `package-install-file`, or via the Package Menu, creates a subdirectory of `package-user-dir` named *`name-version`*, where *name* is the package's name and *version* its version (e.g., `~/.emacs.d/elpa/auctex-11.86/`). We

call this the package's *content directory*. It is where Emacs puts the package's contents (the single Lisp file for a simple package, or the files extracted from a multi-file package).

Emacs then searches every Lisp file in the content directory for autoload magic comments (see Section 16.5 [Autoload], page 305). These autoload definitions are saved to a file named *name*`-autoloads.el` in the content directory. They are typically used to autoload the principal user commands defined in the package, but they can also perform other tasks, such as adding an element to `auto-mode-alist` (see Section 24.2.2 [Auto Major Mode], page 532). Note that a package typically does *not* autoload every function and variable defined within it—only the handful of commands typically called to begin using the package. Emacs then byte-compiles every Lisp file in the package.

After installation, the installed package is *loaded*: Emacs adds the package's content directory to `load-path`, and evaluates the autoload definitions in *name*`-autoloads.el`.

Whenever Emacs starts up, it automatically calls the function `package-activate-all` to make installed packages available to the current session. This is done after loading the early init file, but before loading the regular init file (see Section 42.1.1 [Startup Summary], page 1260). Packages are not automatically made available if the user option `package-enable-at-startup` is set to `nil` in the early init file.

`package-activate-all` [Function]

This function makes the packages available to the current session. The user option `package-load-list` specifies which packages to make available; by default, all installed packages are made available. See Section "Package Installation" in *The GNU Emacs Manual*.

In most cases, you should not need to call `package-activate-all`, as this is done automatically during startup. Simply make sure to put any code that should run before `package-activate-all` in the early init file, and any code that should run after it in the primary init file (see Section "Init File" in *The GNU Emacs Manual*).

`package-initialize` **&optional** *no-activate* [Command]

This function initializes Emacs' internal record of which packages are installed, and then calls `package-activate-all`.

The optional argument *no-activate*, if non-`nil`, causes Emacs to update its record of installed packages without actually making them available.

43.2 Simple Packages

A simple package consists of a single Emacs Lisp source file. The file must conform to the Emacs Lisp library header conventions (see Section D.8 [Library Headers], page 1348). The package's attributes are taken from the various headers, as illustrated by the following example:

```
;;; superfrobnicator.el --- Frobnicate and bifurcate flanges  -*- lexical-binding:t -*

;; Copyright (C) 2022 Free Software Foundation, Inc.

;; Author: J. R. Hacker <jrh@example.com>
;; Version: 1.3
```

```
;; Package-Requires: ((flange "1.0"))
;; Keywords: multimedia, hypermedia
;; URL: https://example.com/jrhacker/superfrobnicate

...

;;; Commentary:

;; This package provides a minor mode to frobnicate and/or
;; bifurcate any flanges you desire.  To activate it, just type
...

;;;###autoload
(define-minor-mode superfrobnicator-mode
...
```

The name of the package is the same as the base name of the file, as written on the first line. Here, it is '`superfrobnicator`'.

The brief description is also taken from the first line. Here, it is '`Frobnicate and bifurcate flanges`'.

The version number comes from the '`Package-Version`' header, if it exists, or from the '`Version`' header otherwise. One or the other *must* be present. Here, the version number is 1.3.

If the file has a '`;;; Commentary:`' section, this section is used as the long description. (When displaying the description, Emacs omits the '`;;; Commentary:`' line, as well as the leading comment characters in the commentary itself.)

If the file has a '`Package-Requires`' header, that is used as the package dependencies. In the above example, the package depends on the '`flange`' package, version 1.0 or higher. See Section D.8 [Library Headers], page 1348, for a description of the '`Package-Requires`' header. If the header is omitted, the package has no dependencies.

The '`Keywords`' and '`URL`' headers are optional, but recommended. The command `describe-package` uses these to add links to its output. The '`Keywords`' header should contain at least one standard keyword from the `finder-known-keywords` list.

The file ought to also contain one or more autoload magic comments, as explained in Section 43.1 [Packaging Basics], page 1308. In the above example, a magic comment autoloads `superfrobnicator-mode`.

See Section 43.4 [Package Archives], page 1312, for an explanation of how to add a single-file package to a package archive.

43.3 Multi-file Packages

A multi-file package is less convenient to create than a single-file package, but it offers more features: it can include multiple Emacs Lisp files, an Info manual, and other file types (such as images).

Prior to installation, a multi-file package is stored in a package archive as a tar file. The tar file must be named *`name-version`*`.tar`, where *name* is the package name and *version*

is the version number. Its contents, once extracted, must all appear in a directory named *name*`-`*version*, the *content directory* (see Section 43.1 [Packaging Basics], page 1308). Files may also extract into subdirectories of the content directory.

One of the files in the content directory must be named *name*`-pkg.el`. It must contain a single Lisp form, consisting of a call to the function `define-package`, described below. This defines the package's attributes: version, brief description, and requirements.

For example, if we distribute version 1.3 of the superfrobnicator as a multi-file package, the tar file would be `superfrobnicator-1.3.tar`. Its contents would extract into the directory `superfrobnicator-1.3`, and one of these would be the file `superfrobnicator-pkg.el`.

`define-package` *name version* **&optional** *docstring requirements* [Function]
: This function defines a package. *name* is the package name, a string. *version* is the version, as a string of a form that can be understood by the function `version-to-list`. *docstring* is the brief description.

  requirements is a list of required packages and their versions. Each element in this list should have the form (*dep-name dep-version*), where *dep-name* is a symbol whose name is the dependency's package name, and *dep-version* is the dependency's version (a string).

If the content directory contains a file named `README`, this file is used as the long description (overriding any '`;;; Commentary:`' section).

If the content directory contains a file named `dir`, this is assumed to be an Info directory file made with `install-info`. See Section "Invoking install-info" in *Texinfo*. The relevant Info files should also be present in the content directory. In this case, Emacs will automatically add the content directory to `Info-directory-list` when the package is activated.

Do not include any `.elc` files in the package. Those are created when the package is installed. Note that there is no way to control the order in which files are byte-compiled.

Do not include any file named *name*`-autoloads.el`. This file is reserved for the package's autoload definitions (see Section 43.1 [Packaging Basics], page 1308). It is created automatically when the package is installed, by searching all the Lisp files in the package for autoload magic comments.

If the multi-file package contains auxiliary data files (such as images), the package's Lisp code can refer to these files via the variable `load-file-name` (see Chapter 16 [Loading], page 299). Here is an example:

```
(defconst superfrobnicator-base (file-name-directory load-file-name))

(defun superfrobnicator-fetch-image (file)
  (expand-file-name file superfrobnicator-base))
```

If your package contains files that you don't wish to distribute to users (e.g. regression tests), you can add them to an `.elpaignore` file. In this file, each line lists a file or a wildcard matching files; those files should be ignored when producing your package's tarball on ELPA (see Section 43.4 [Package Archives], page 1312). (ELPA will pass this file to the `tar` command via the `-X` command-line option, when it prepares the package for download.)

43.4 Creating and Maintaining Package Archives

Via the Package Menu, users may download packages from *package archives*. Such archives are specified by the variable `package-archives`, whose default value lists the archives hosted on GNU ELPA (`https://elpa.gnu.org`) and non-GNU ELPA (`https://elpa.nongnu.org`). This section describes how to set up and maintain a package archive.

`package-archives` [User Option]

The value of this variable is an alist of package archives recognized by the Emacs package manager.

Each alist element corresponds to one archive, and should have the form `(`*`id`*` . `*`location`*`)`, where *id* is the name of the archive (a string) and *location* is its *base location* (a string).

If the base location starts with '`http:`' or '`https:`', it is treated as an HTTP(S) URL, and packages are downloaded from this archive via HTTP(S) (as is the case for the default GNU archive).

Otherwise, the base location should be a directory name. In this case, Emacs retrieves packages from this archive via ordinary file access. Such local archives are mainly useful for testing.

A package archive is simply a directory in which the package files, and associated files, are stored. If you want the archive to be reachable via HTTP, this directory must be accessible to a web server; See Section 43.5 [Archive Web Server], page 1313.

A convenient way to set up and update a package archive is via the `package-x` library. This is included with Emacs, but not loaded by default; type *`M-x load-library`* `RET` *`package-x`* `RET` to load it, or add `(require 'package-x)` to your init file. See Section "Lisp Libraries" in *The GNU Emacs Manual*.

After you create an archive, remember that it is not accessible in the Package Menu interface unless it is in `package-archives`.

Maintaining a public package archive entails a degree of responsibility. When Emacs users install packages from your archive, those packages can cause Emacs to run arbitrary code with the permissions of the installing user. (This is true for Emacs code in general, not just for packages.) So you should ensure that your archive is well-maintained and keep the hosting system secure.

One way to increase the security of your packages is to *sign* them using a cryptographic key. If you have generated a private/public gpg key pair, you can use gpg to sign the package like this:

```
gpg -ba -o file.sig file
```

For a single-file package, *file* is the package Lisp file; for a multi-file package, it is the package tar file. You can also sign the archive's contents file in the same way. Make the `.sig` files available in the same location as the packages. You should also make your public key available for people to download; e.g., by uploading it to a key server such as `https://pgp.mit.edu/`. When people install packages from your archive, they can use your public key to verify the signatures.

A full explanation of these matters is outside the scope of this manual. For more information on cryptographic keys and signing, see Section "GnuPG" in *The GNU Privacy Guard*

Manual. Emacs comes with an interface to GNU Privacy Guard, see Section "EasyPG" in *Emacs EasyPG Assistant Manual*.

43.5 Interfacing to an archive web server

A web server providing access to a package archive must support the following queries:

archive-contents
: Return a lisp form describing the archive contents. The form is a list of 'package-desc' structures (see `package.el`), except the first element of the list is the archive version.

<package name>-readme.txt
: Return the long description of the package.

<file name>.sig
: Return the signature for the file.

<file name>
: Return the file. This will be the tarball for a multi-file package, or the single file for a simple package.

Appendix A Emacs 28 Antinews

For those users who live backwards in time, here is information about downgrading to Emacs version 28.2. We hope you will enjoy the greater simplicity that results from the absence of many Emacs 29.1.90 features.

- The implementation of overlays is back to its simple, time-proven storage in a pair of linear linked lists centered around some buffer position. No more fancy interval trees and suchlikes. Lisp programs that use overlays once again need to recenter overlays around the buffer position of interest, and display-related features should again make sure they don't use too many overlays in a buffer, lest redisplay will be too slow.
- Several functions stopped the annoying conversion of quotes and key sequences by no longer calling `substitute-command-keys`. One prominent example is `format-prompt` and all its many callers. This makes the strings they produce much more predictable, returning to you, the Lisp programmer, control on which punctuation characters will appear in the text presented to the users. For similar reasons, the `substitute-quotes` function was deleted.
- The venerable `buffer-modified-p` function again reliably returns either `nil` or `t`, not any other confusing values.
- The support for '`medium`' weight of fonts was dropped. Emacs now considers '`medium`' and '`regular`' weights to be the same. We believe this will simplify your font setup, since there's no longer a need to worry about fonts that support '`regular`' weight, but not the '`medium`' one, or vice versa: either one will do!
- To reduce the amount of code in Emacs related to unimportant features, we've removed the function `compiled-function-p`. Lisp programs are expected to test explicitly for the relevant types of function objects: built-in, byte-compiled, and natively-compiled. For the same reasons we deleted the functions `pos-bol`, `pos-eol`, `file-attribute-file-identifier`, and quite a few others. We don't expect anyone to miss those fancy functions.
- The timeout used by `x-show-tip` can no longer be specified by Lisp programs; it is hard-coded in the function. This will lead to a simpler, easier maintained code, and no one should want to control the timeout after which the tip pops down.
- The macro `setopt` was deleted; use `customize-variable` instead, or invoke the `:set` function from Lisp.
- We removed the `lisp-directory` variable, as the value can be easily deduced from other similar variables, like `installation-directory` and `source-directory`, each one when it's relevant.
- To simplify code and reduce complexity, we deleted the functions `get-display-property` and `add-display-text-property`; use the generic `get-text-property` and `put-text-property` instead.
- Support for pinch input events and for modern drag-and-drop functionality on X was dropped. As you move back in time, these facilities will become less and less important, and will soon enough disappear, so there's no reason to keep them in Emacs.
- To keep Emacs clean and elegant, we've removed the `textsec.el` library, with its facilities for checking whether some text is "suspicious". We consider our users smart

enough to detect maliciously modified text by just looking at it or by moving the cursor across it, and the whole idea that someone would wish to deliberately deceive Emacs users ridiculous and unworthy of complicating our elegant text-processing and display capabilities.

- The functions `keymap-set`, `keymap-global-set`, `keymap-local-set`, `keymap-substitute`, `keymap-lookup`, and some others were deleted. We have found the traditional `define-key`, `global-set-key`, `local-set-key`, `substitute-key-definition`, and `key-binding` more than enough, and their minor inconsistencies in the syntax of keys they accept a source of endless fun in Emacs Lisp programming. Why make Emacs programming a dull place? For the same reasons we deleted `key-valid-p`, since we consider the permissive nature of `kbd` more in the spirit of Emacs Lisp.
- Yanking of anything but plain text from other applications becomes more and more an unnecessary feature as you move back in time, so we dropped support for pasting media like HTML and images via the clipboard. If you *really* need to yank those into an Emacs buffer, you can go via a disk file.
- We removed unnecessary functions `string-pixel-width` and `string-glyph-split`, as we consider it inappropriate for Lisp programs to do display layout calculations, where these functions come in handy. Display is for the display engine, written in C, and should stay there!
- Various new Xwidget functions, such as `xwidget-perform-lispy-event`, `xwidget-webkit-load-html`, and `xwidget-webkit-back-forward-list`, were deleted as part of our continuing effort to gradually delete the entire Xwidget functionality in some previous release of Emacs.
- Setting the `:stderr` property of a process in a `make-process` call once again forces the process's connection to use pipes, not ptys, for all the standard streams — a considerable simplification of this complex interface.
- To keep the amount of Lisp functions from growing out of control, we deleted `string-equal-ignore-case`. Use `compare-strings` instead.

  Several features that complicated the byte compiler have been removed:
  - The warnings about quoting mistakes in documentation strings. You are expected to find such mistakes yourself, by eyeballing the resulting `*Help*` buffer display.
  - The warnings about malformed `defcustom` types, like double-quoting symbols in `choice` lists.
- The macro `with-buffer-unmodified-if-unchanged` was deleted. Lisp programs that need to leave the buffer unmodified in these cases can always compare the text before and after the modifications.
- The functions `string-edit` and `read-string-from-buffer` were removed, as we consider the fun of programming them anew every time an important part of the education of each Emacs Lisp developer.
- We deleted the function `readablep` and the related variable `print-unreadable-function`, since no one is supposed to want to print unreadable Lisp objects.
- The facility for storing multisession variables was deleted as an unnecessary complication. With it are gone `multisession-value`, `define-multisession-variable`, and `list-multisession-values`.

- The support for the `cursor-face` text property was dropped. We consider the rest of the faces adequate for supporting this functionality.
- The function `tooltip-show` dropped support for optional face arguments `text-face` and `default-face` that allow fancy control of the face of the tip text and top frame colors. We decided that tooltips should all look the same, to prevent user confusion.
- As part of the ongoing quest for simplicity, many other functions and variables have been eliminated. Other functions and variables, that were declared obsolete since Emacs 24, have been added back, in preparation for releasing Emacs 24 in some distant past.

Appendix B GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.
`https://fsf.org/`

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released

under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any,

be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their

titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `https://www.gnu.org/licenses/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

    "Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    "CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

    "Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

    An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

    The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Appendix C GNU General Public License

Version 3, 29 June 2007

Copyright © 2007 Free Software Foundation, Inc. https://fsf.org/

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program—to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions.

   "This License" refers to version 3 of the GNU General Public License.

   "Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

   "The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

   To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

   A "covered work" means either the unmodified Program or a work based on the Program.

   To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

   To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

   An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

   The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

   A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

   All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

   You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

   Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

   No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

   You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

   You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

   You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

   a. The work must carry prominent notices stating that you modified it, and giving a relevant date.

   b. The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".

   c. You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

   d. If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

   A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

   You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

a. Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

b. Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

c. Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

d. Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

e. Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source.

The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

a. Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

b. Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

c. Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

d. Limiting the use for publicity purposes of names of licensors or authors of the material; or

e. Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

f. Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance.

However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so

available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

    If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

    Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) year name of author

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or (at
your option) any later version.

This program is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see https://www.gnu.org/licenses/.
```

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
program Copyright (C) year name of author
This program comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’.
This is free software, and you are welcome to redistribute it
under certain conditions; type ‘show c’ for details.
```

The hypothetical commands ‘`show w`’ and ‘`show c`’ should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see `https://www.gnu.org/licenses/`.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read `https://www.gnu.org/licenses/why-not-lgpl.html`.

Appendix D Tips and Conventions

This chapter describes no additional features of Emacs Lisp. Instead it gives advice on making effective use of the features described in the previous chapters, and describes conventions Emacs Lisp programmers should follow.

You can automatically check some of the conventions described below by running the command *M-x checkdoc* RET when visiting a Lisp file. It cannot check all of the conventions, and not all the warnings it gives necessarily correspond to problems, but it is worth examining them all. Alternatively, use the command *M-x checkdoc-current-buffer* RET to check the conventions in the current buffer, or `checkdoc-file` when you want to check a file in batch mode, e.g., with a command run by *M-x compile* RET.

D.1 Emacs Lisp Coding Conventions

Here are conventions that you should follow when writing Emacs Lisp code intended for widespread use:

- Simply loading a package should not change Emacs's editing behavior. Include a command or commands to enable and disable the feature, or to invoke it.

  This convention is mandatory for any file that includes custom definitions. If fixing such a file to follow this convention requires an incompatible change, go ahead and make the incompatible change; don't postpone it.

- You should choose a short word to distinguish your program from other Lisp programs. The names of all global symbols in your program, that is the names of variables, constants, and functions, should begin with that chosen prefix. Separate the prefix from the rest of the name with a hyphen, '`-`'. This practice helps avoid name conflicts, since all global variables in Emacs Lisp share the same name space, and all functions share another name space[1]. Use two hyphens to separate prefix and name if the symbol is not meant to be used by other packages.

  Occasionally, for a command name intended for users to use, it is more convenient if some words come before the package's name prefix. For example, it is our convention to have commands that list objects named as '`list-`*`something`*', e.g., a package called '`frob`' could have a command '`list-frobs`', when its other global symbols begin with '`frob-`'. Also, constructs that define functions, variables, etc., work better if they start with '`define-`', so put the name prefix later on in the name.

  This recommendation applies even to names for traditional Lisp primitives that are not primitives in Emacs Lisp—such as `copy-list`. Believe it or not, there is more than one plausible way to define `copy-list`. Play it safe; append your name prefix to produce a name like `foo-copy-list` or `mylib-copy-list` instead.

  If you write a function that you think ought to be added to Emacs under a certain name, such as `twiddle-files`, don't call it by that name in your program. Call it `mylib-twiddle-files` in your program, and send mail to '`bug-gnu-emacs@gnu.org`' suggesting we add it to Emacs. If and when we do, we can change the name easily enough.

[1] The benefits of a Common Lisp-style package system are considered not to outweigh the costs.

If one prefix is insufficient, your package can use two or three alternative common prefixes, so long as they make sense.

- We recommend enabling `lexical-binding` in new code, and converting existing Emacs Lisp code to enable `lexical-binding` if it doesn't already. See Section 12.10.4 [Using Lexical Binding], page 204.
- Put a call to `provide` at the end of each separate Lisp file. See Section 16.7 [Named Features], page 310.
- If a file requires certain other Lisp programs to be loaded beforehand, then the comments at the beginning of the file should say so. Also, use `require` to make sure they are loaded. See Section 16.7 [Named Features], page 310.
- If a file *foo* uses a macro defined in another file *bar*, but does not use any functions or variables defined in *bar*, then *foo* should contain the following expression:

  ```
  (eval-when-compile (require 'bar))
  ```

  This tells Emacs to load *bar* just before byte-compiling *foo*, so that the macro definition is available during compilation. Using `eval-when-compile` avoids loading *bar* when the compiled version of *foo* is *used*. It should be called before the first use of the macro in the file. See Section 14.3 [Compiling Macros], page 271.
- Avoid loading additional libraries at run time unless they are really needed. If your file simply cannot work without some other library, then just `require` that library at the top-level and be done with it. But if your file contains several independent features, and only one or two require the extra library, then consider putting `require` statements inside the relevant functions rather than at the top-level. Or use `autoload` statements to load the extra library when needed. This way people who don't use those aspects of your file do not need to load the extra library.
- If you need Common Lisp extensions, use the `cl-lib` library rather than the old `cl` library. The latter library is deprecated and will be removed in a future version of Emacs.
- When defining a major mode, please follow the major mode conventions. See Section 24.2.1 [Major Mode Conventions], page 529.
- When defining a minor mode, please follow the minor mode conventions. See Section 24.3.1 [Minor Mode Conventions], page 545.
- If the purpose of a function is to tell you whether a certain condition is true or false, give the function a name that ends in '`p`' (which stands for "predicate"). If the name is one word, add just '`p`'; if the name is multiple words, add '`-p`'. Examples are `framep` and `frame-live-p`. We recommend to avoid using this `-p` suffix in boolean variable names, unless the variable is bound to a predicate function; instead, use a `-flag` suffix or names like `is-foo`.
- If the purpose of a variable is to store a single function, give it a name that ends in '`-function`'. If the purpose of a variable is to store a list of functions (i.e., the variable is a hook), please follow the naming conventions for hooks. See Section 24.1 [Hooks], page 525.
- Using `unload-feature` will undo the changes usually done by loading a feature (like adding functions to hooks). However, if loading *feature* does something unusual and more complex, you can define a function named *`feature`*`-unload-function`, and make

it undo any such special changes. `unload-feature` will then automatically run this function if it exists. See Section 16.9 [Unloading], page 314.

- It is a bad idea to define aliases for the Emacs primitives. Normally you should use the standard names instead. The case where an alias may be useful is where it facilitates backwards compatibility or portability.
- If a package needs to define an alias or a new function for compatibility with some other version of Emacs, name it with the package prefix, not with the raw name with which it occurs in the other version. Here is an example from Gnus, which provides many examples of such compatibility issues.

```
(defalias 'gnus-point-at-bol
  (if (fboundp 'point-at-bol)
      'point-at-bol
    'line-beginning-position))
```

- Redefining or advising an Emacs primitive is a bad idea. It may do the right thing for a particular program, but there is no telling what other programs might break as a result.
- It is likewise a bad idea for one Lisp package to advise a function in another Lisp package (see Section 13.12 [Advising Functions], page 252).
- Avoid using `eval-after-load` and `with-eval-after-load` in libraries and packages (see Section 16.10 [Hooks for Loading], page 314). This feature is meant for personal customizations; using it in a Lisp program is unclean, because it modifies the behavior of another Lisp file in a way that's not visible in that file. This is an obstacle for debugging, much like advising a function in the other package.
- If a file does replace any of the standard functions or library programs of Emacs, prominent comments at the beginning of the file should say which functions are replaced, and how the behavior of the replacements differs from that of the originals.
- Constructs that define a function or variable should be macros, not functions, and their names should start with '`define-`'. The macro should receive the name to be defined as the first argument. That will help various tools find the definition automatically. Avoid constructing the names in the macro itself, since that would confuse these tools.
- In some other systems there is a convention of choosing variable names that begin and end with '`*`'. We don't use that convention in Emacs Lisp, so please don't use it in your programs. (Emacs uses such names only for special-purpose buffers.) People will find Emacs more coherent if all libraries use the same conventions.
- The default file coding system for Emacs Lisp source files is UTF-8 (see Section 34.1 [Text Representations], page 966). In the rare event that your program contains characters which are *not* in UTF-8, you should specify an appropriate coding system in the source file's '`-*-`' line or local variables list. See Section "Local Variables in Files" in *The GNU Emacs Manual*.
- Indent the file using the default indentation parameters.
- Don't make a habit of putting close-parentheses on lines by themselves; Lisp programmers find this disconcerting.
- Please put a copyright notice and copying permission notice on the file if you distribute copies. See Section D.8 [Library Headers], page 1348.

- For variables holding (or functions returning) a file or directory name, avoid using `path` in its name, preferring `file`, `file-name`, or `directory` instead, since Emacs follows the GNU convention to use the term *path* only for search paths, which are lists of directory names.

D.2 Key Binding Conventions

- Many special major modes, like Dired, Info, Compilation, and Occur, are designed to handle read-only text that contains *hyper-links*. Such a major mode should redefine *`mouse-2`* and `RET` to follow the links. It should also set up a `follow-link` condition, so that the link obeys `mouse-1-click-follows-link`. See Section 33.19.8 [Clickable Text], page 937. See Section 41.20 [Buttons], page 1236, for an easy method of implementing such clickable links.
- Don't define *`C-c letter`* as a key in Lisp programs. Sequences consisting of *`C-c`* and a letter (either upper or lower case; ASCII or non-ASCII) are reserved for users; they are the **only** sequences reserved for users, so do not block them.

  Changing all the Emacs major modes to respect this convention was a lot of work; abandoning this convention would make that work go to waste, and inconvenience users. Please comply with it.
- Function keys `F5` through `F9` without modifier keys are also reserved for users to define.
- Sequences consisting of *`C-c`* followed by a control character or a digit are reserved for major modes.
- Sequences consisting of *`C-c`* followed by *`{`*, *`}`*, `<`, `>`, *`:`* or *`;`* are also reserved for major modes.
- Sequences consisting of *`C-c`* followed by any other ASCII punctuation or symbol character are allocated for minor modes. Using them in a major mode is not absolutely prohibited, but if you do that, the major mode binding may be shadowed from time to time by minor modes.
- Don't bind *`C-h`* following any prefix character (including *`C-c`*). If you don't bind *`C-h`*, it is automatically available as a help character for listing the subcommands of the prefix character.
- Don't bind a key sequence ending in `ESC` except following another `ESC`. (That is, it is OK to bind a sequence ending in `ESC ESC`.)

  The reason for this rule is that a non-prefix binding for `ESC` in any context prevents recognition of escape sequences as function keys in that context.
- Similarly, don't bind a key sequence ending in *`C-g`*, since that is commonly used to cancel a key sequence.
- Anything that acts like a temporary mode or state that the user can enter and leave should define `ESC ESC` or `ESC ESC ESC` as a way to escape.

  For a state that accepts ordinary Emacs commands, or more generally any kind of state in which `ESC` followed by a function key or arrow key is potentially meaningful, then you must not define `ESC ESC`, since that would preclude recognizing an escape sequence after `ESC`. In these states, you should define `ESC ESC ESC` as the way to escape. Otherwise, define `ESC ESC` instead.

D.3 Emacs Programming Tips

Following these conventions will make your program fit better into Emacs when it runs.

- Don't use `next-line` or `previous-line` in programs; nearly always, `forward-line` is more convenient as well as more predictable and robust. See Section 31.2.4 [Text Lines], page 858.
- Don't call functions that set the mark, unless setting the mark is one of the intended features of your program. The mark is a user-level feature, so it is incorrect to change the mark except to supply a value for the user's benefit. See Section 32.7 [The Mark], page 875.

  In particular, don't use any of these functions:

  - `beginning-of-buffer`, `end-of-buffer`
  - `replace-string`, `replace-regexp`
  - `insert-file`, `insert-buffer`

  If you just want to move point, or replace a certain string, or insert a file or buffer's contents, without any of the other features intended for interactive users, you can replace these functions with one or two lines of simple Lisp code.
- Use lists rather than vectors, except when there is a particular reason to use a vector. Lisp has more facilities for manipulating lists than for vectors, and working with lists is usually more convenient.

  Vectors are advantageous for tables that are substantial in size and are accessed in random order (not searched front to back), provided there is no need to insert or delete elements (only lists allow that).
- The recommended way to show a message in the echo area is with the `message` function, not `princ`. See Section 41.4 [The Echo Area], page 1135.
- When you encounter an error condition, call the function `error` (or `signal`). The function `error` does not return. See Section 11.7.3.1 [Signaling Errors], page 180.

  Don't use `message`, `throw`, `sleep-for`, or `beep` to report errors.
- An error message should start with a capital letter but should not end with a period or other punctuation.

  It is occasionally useful to tell the user where an error originated, even if `debug-on-error` is `nil`. In such cases, a lower-case Lisp symbol can be prepended to the error message. For example, the error message "Invalid input" could be extended to say "some-function: Invalid input".
- A question asked in the minibuffer with `yes-or-no-p` or `y-or-n-p` should start with a capital letter and end with '`?`'.
- When you mention a default value in a minibuffer prompt, put it and the word '`default`' inside parentheses. It should look like this:

  ```
  Enter the answer (default 42):
  ```
- In `interactive`, if you use a Lisp expression to produce a list of arguments, don't try to provide the correct default values for region or position arguments. Instead, provide `nil` for those arguments if they were not specified, and have the function body compute the default value when the argument is `nil`. For instance, write this:

  ```
  (defun foo (pos)
  ```

```
(interactive
 (list (if specified specified-pos)))
(unless pos (setq pos default-pos))
...)
```

rather than this:

```
(defun foo (pos)
  (interactive
   (list (if specified specified-pos
             default-pos)))
  ...)
```

This is so that repetition of the command will recompute these defaults based on the current circumstances.

You do not need to take such precautions when you use interactive specs '`d`', '`m`' and '`r`', because they make special arrangements to recompute the argument values on repetition of the command.

- Many commands that take a long time to execute display a message that says something like '`Operating...`' when they start, and change it to '`Operating...done`' when they finish. Please keep the style of these messages uniform: *no* space around the ellipsis, and *no* period after '`done`'. See Section 41.4.2 [Progress], page 1138, for an easy way to generate such messages.
- Try to avoid using recursive edits. Instead, do what the Rmail `e` command does: use a new local keymap that contains a command defined to switch back to the old local keymap. Or simply switch to another buffer and let the user switch back at will. See Section 22.13 [Recursive Editing], page 476.

D.4 Tips for Making Compiled Code Fast

Here are ways of improving the execution speed of byte-compiled Lisp programs.

- Profile your program, to find out where the time is being spent. See Section 19.5 [Profiling], page 370.
- Use iteration rather than recursion whenever possible. Function calls are slow in Emacs Lisp even when a compiled function is calling another compiled function.
- Using the primitive list-searching functions `memq`, `member`, `assq`, or `assoc` is even faster than explicit iteration. It can be worth rearranging a data structure so that one of these primitive search functions can be used.
- Certain built-in functions are handled specially in byte-compiled code, avoiding the need for an ordinary function call. It is a good idea to use these functions rather than alternatives. To see whether a function is handled specially by the compiler, examine its `byte-compile` property. If the property is non-`nil`, then the function is handled specially.

  For example, the following input will show you that `aref` is compiled specially (see Section 6.3 [Array Functions], page 116):

```
(get 'aref 'byte-compile)
     ⇒ byte-compile-two-args
```

Note that in this case (and many others), you must first load the `bytecomp` library, which defines the `byte-compile` property.

- If calling a small function accounts for a substantial part of your program's running time, make the function inline. This eliminates the function call overhead. Since making a function inline reduces the flexibility of changing the program, don't do it unless it gives a noticeable speedup in something slow enough that users care about the speed. See Section 13.14 [Inline Functions], page 261.

D.5 Tips for Avoiding Compiler Warnings

- Try to avoid compiler warnings about undefined free variables, by adding dummy `defvar` definitions for these variables, like this:

  ```
  (defvar foo)
  ```

  Such a definition has no effect except to tell the compiler not to warn about uses of the variable `foo` in this file.

- Similarly, to avoid a compiler warning about an undefined function that you know *will* be defined, use a `declare-function` statement (see Section 13.16 [Declaring Functions], page 265).

- If you use many functions, macros, and variables from a certain file, you can add a `require` (see Section 16.7 [Named Features], page 310) for that package to avoid compilation warnings for them, like this:

  ```
  (require 'foo)
  ```

  If you need only macros from some file, you can require it only at compile time (see Section 17.5 [Eval During Compile], page 321). For instance,

  ```
  (eval-when-compile
    (require 'foo))
  ```

- If you bind a variable in one function, and use it or set it in another function, the compiler warns about the latter function unless the variable has a definition. But adding a definition would be unclean if the variable has a short name, since Lisp packages should not define short variable names. The right thing to do is to rename this variable to start with the name prefix used for the other functions and variables in your package.

- The last resort for avoiding a warning, when you want to do something that is usually a mistake but you know is not a mistake in your usage, is to put it inside `with-no-warnings`. See Section 17.6 [Compiler Errors], page 322.

D.6 Tips for Documentation Strings

Here are some tips and conventions for the writing of documentation strings. You can check many of these conventions by running the command *`M-x checkdoc-minor-mode`*.

- Every command, function, or variable intended for users to know about should have a documentation string.

- An internal variable or subroutine of a Lisp program might as well have a documentation string. Documentation strings take up very little space in a running Emacs.

- Format the documentation string so that it fits in an Emacs window on an 80-column screen. It is a good idea for most lines to be no wider than 60 characters. The first line should not be wider than 67 characters or it will look bad in the output of `apropos`.

  You can fill the text if that looks good. Emacs Lisp mode fills documentation strings to the width specified by `emacs-lisp-docstring-fill-column`. However, you can sometimes make a documentation string much more readable by adjusting its line breaks with care. Use blank lines between sections if the documentation string is long.

- The first line of the documentation string should consist of one or two complete sentences that stand on their own as a summary. *`M-x apropos`* displays just the first line, and if that line's contents don't stand on their own, the result looks bad. In particular, start the first line with a capital letter and end it with a period.

  For a function, the first line should briefly answer the question, "What does this function do?" For a variable, the first line should briefly answer the question, "What does this value mean?" Prefer to answer these questions in a way that will make sense to users and callers of the function or the variable. In particular, do *not* tell what the function does by enumerating the actions of its code; instead, describe the role of these actions and the function's contract.

  Don't limit the documentation string to one line; use as many lines as you need to explain the details of how to use the function or variable. Please use complete sentences for the rest of the text too.

- When the user tries to use a disabled command, Emacs displays just the first paragraph of its documentation string—everything through the first blank line. If you wish, you can choose which information to include before the first blank line so as to make this display useful.

- The first line should mention all the important arguments of the function (in particular, the mandatory arguments), and should mention them in the order that they are written in a function call. If the function has many arguments, then it is not feasible to mention them all in the first line; in that case, the first line should mention the first few arguments, including the most important arguments.

- When a function's documentation string mentions the value of an argument of the function, use the argument name in capital letters as if it were a name for that value. Thus, the documentation string of the function `eval` refers to its first argument as '`FORM`', because the actual argument name is `form`:

  ```
  Evaluate FORM and return its value.
  ```

  Also write metasyntactic variables in capital letters, such as when you show the decomposition of a list or vector into subunits, some of which may vary. '`KEY`' and '`VALUE`' in the following example illustrate this practice:

  ```
  The argument TABLE should be an alist whose elements
  have the form (KEY . VALUE).  Here, KEY is ...
  ```

- Never change the case of a Lisp symbol when you mention it in a doc string. If the symbol's name is `foo`, write "foo", not "Foo" (which is a different symbol).

  This might appear to contradict the policy of writing function argument values, but there is no real contradiction; the argument *value* is not the same thing as the *symbol* that the function uses to hold the value.

If this puts a lower-case letter at the beginning of a sentence and that annoys you, rewrite the sentence so that the symbol is not at the start of it.

- Do not start or end a documentation string with whitespace.
- **Do not** indent subsequent lines of a documentation string so that the text is lined up in the source code with the text of the first line. This looks nice in the source code, but looks bizarre when users view the documentation. Remember that the indentation before the starting double-quote is not part of the string!
- When documentation should display an ASCII apostrophe or grave accent, use '`\\='` or '`\\=`' in the documentation string literal so that the character is displayed as-is.
- In documentation strings, do not quote expressions that are not Lisp symbols, as these expressions can stand for themselves. For example, write '`Return the list (NAME TYPE RANGE) ...`' instead of '`Return the list `(NAME TYPE RANGE)' ...`' or '`Return the list \\='(NAME TYPE RANGE) ...`'.
- When a documentation string refers to a Lisp symbol, write it as it would be printed (which usually means in lower case), with a grave accent '`` ` `` ' before and apostrophe '`'`' after it. There are two exceptions: write `t` and `nil` without surrounding punctuation. For example:

  ```
  CODE can be `lambda', nil, or t.
  ```

  Note that when Emacs displays these doc strings, Emacs will usually display '`` ` `` ' (grave accent) as '‘' (left single quotation mark) and '`'`' (apostrophe) as '’' (right single quotation mark), if the display supports displaying these characters. See Section 25.3 [Keys in Documentation], page 600. (Some previous versions of this section recommended using the non-ASCII single quotation marks directly in doc strings, but this is now discouraged, since that leads to broken help string displays on terminals that don't support displaying those characters.)

  Help mode automatically creates a hyperlink when a documentation string uses a single-quoted symbol name, if the symbol has either a function or a variable definition. You do not need to do anything special to make use of this feature. However, when a symbol has both a function definition and a variable definition, and you want to refer to just one of them, you can specify which one by writing one of the words '`variable`', '`option`', '`function`', or '`command`', immediately before the symbol name. (Case makes no difference in recognizing these indicator words.) For example, if you write

  ```
  This function sets the variable `buffer-file-name'.
  ```

  then the hyperlink will refer only to the variable documentation of `buffer-file-name`, and not to its function documentation.

  If a symbol has a function definition and/or a variable definition, but those are irrelevant to the use of the symbol that you are documenting, you can write the words '`symbol`' or '`program`' before the symbol name to prevent making any hyperlink. For example,

  ```
  If the argument KIND-OF-RESULT is the symbol `list',
  this function returns a list of all the objects
  that satisfy the criterion.
  ```

  does not make a hyperlink to the documentation, irrelevant here, of the function `list`.

Normally, no hyperlink is made for a variable without variable documentation. You can force a hyperlink for such variables by preceding them with one of the words '`variable`' or '`option`'.

Hyperlinks for faces are only made if the face name is preceded or followed by the word '`face`'. In that case, only the face documentation will be shown, even if the symbol is also defined as a variable or as a function.

To make a hyperlink to Info documentation, write the single-quoted name of the Info node (or anchor), preceded by '`info node`', '`Info node`', '`info anchor`' or '`Info anchor`'. The Info file name defaults to '`emacs`'. For example,

```
See Info node `Font Lock' and Info node `(elisp)Font Lock Basics'.
```

To make a hyperlink to a man page, write the single-quoted name of the man page, preceded by '`Man page`', '`man page`', or '`man page for`'. For example,

```
See the man page `chmod(1)' for details.
```

The Info documentation is always preferable to man pages, so be sure to link to an Info manual where available. For example, `chmod` is documented in the GNU Coreutils manual, so it is better to link to that instead of the man page.

To link to a customization group, write the single-quoted name of the group, preceded by '`customization group`' (the first character in each word is case-insensitive). For example,

```
See the customization group `whitespace' for details.
```

Finally, to create a hyperlink to URLs, write the single-quoted URL, preceded by '`URL`'. For example,

```
The GNU project website has more information (see URL
`https://www.gnu.org/').
```

- Don't write key sequences directly in documentation strings. Instead, use the '`\\[...]`' construct to stand for them. For example, instead of writing '`C-f`', write the construct '`\\[forward-char]`'. When Emacs displays the documentation string, it substitutes whatever key is currently bound to `forward-char`. (This is normally '`C-f`', but it may be some other character if the user has moved key bindings.) See Section 25.3 [Keys in Documentation], page 600.

- In documentation strings for a major mode, you will want to refer to the key bindings of that mode's local map, rather than global ones. Therefore, use the construct '`\\<...>`' once in the documentation string to specify which key map to use. Do this before the first use of '`\\[...]`', and not in the middle of a sentence (since if the map is not loaded, the reference to the map will be replaced with a sentence saying the map is not currently defined). The text inside the '`\\<...>`' should be the name of the variable containing the local keymap for the major mode.

  Each use of '`\\[...]`' slows the display of the documentation string by a tiny amount. If you use a lot of them, these tiny slowdowns will add up, and might become tangible, especially on slow systems. So our recommendation is not to over-use them; e.g., try to avoid using more than one reference to the same command in the same doc string.

- For consistency, phrase the verb in the first sentence of a function's documentation string as an imperative—for instance, use "Return the cons of A and B." in preference to "Returns the cons of A and B." Usually it looks good to do likewise for the rest

of the first paragraph. Subsequent paragraphs usually look better if each sentence is indicative and has a proper subject.

- The documentation string for a function that is a yes-or-no predicate should start with words such as "Return t if", to indicate explicitly what constitutes truth. The word "return" avoids starting the sentence with lower-case "t", which could be somewhat distracting.
- Write documentation strings in the active voice, not the passive, and in the present tense, not the future. For instance, use "Return a list containing A and B." instead of "A list containing A and B will be returned."
- Avoid using the word "cause" (or its equivalents) unnecessarily. Instead of, "Cause Emacs to display text in boldface", write just "Display text in boldface".
- Avoid using "iff" (a mathematics term meaning "if and only if"), since many people are unfamiliar with it and mistake it for a typo. In most cases, the meaning is clear with just "if". Otherwise, try to find an alternate phrasing that conveys the meaning.
- Try to avoid using abbreviations such as "e.g." (for "for example"), "i.e." (for "that is"), "no." (for "number"), "cf." (for "compare"/"see also") and "w.r.t." (for "with respect to") as much as possible. It is almost always clearer and easier to read the expanded version.[2]
- When a command is meaningful only in a certain mode or situation, do mention that in the documentation string. For example, the documentation of `dired-find-file` is:

  ```
  In Dired, visit the file or directory named on this line.
  ```

- When you define a variable that represents an option users might want to set, use `defcustom`. See Section 12.5 [Defining Variables], page 194.
- The documentation string for a variable that is a yes-or-no flag should start with words such as "Non-nil means", to make it clear that all non-`nil` values are equivalent and indicate explicitly what `nil` and non-`nil` mean.
- If a line in a documentation string begins with an open-parenthesis, consider writing a backslash before the open-parenthesis, like this:

  ```
  The argument FOO can be either a number
  \(a buffer position) or a string (a file name).
  ```

  This avoids a bug in Emacs versions older than 27.1, where the '`(`' was treated as the start of a defun (see Section "Defuns" in *The GNU Emacs Manual*). If you do not anticipate anyone editing your code with older Emacs versions, there is no need for this work-around.

D.7 Tips on Writing Comments

We recommend these conventions for comments:

'`;`' Comments that start with a single semicolon, '`;`', should all be aligned to the same column on the right of the source code. Such comments usually explain how the code on that line does its job. For example:

[2] We do use these occasionally, but try not to overdo it.

```
(setq base-version-list                 ; There was a base
      (assoc (substring fn 0 start-vn)  ; version to which
             file-version-assoc-list))  ; this looks like
                                        ; a subversion.
```

‘;;’ Comments that start with two semicolons, ‘;;’, should be aligned to the same level of indentation as the code. Such comments usually describe the purpose of the following lines or the state of the program at that point. For example:

```
(prog1 (setq auto-fill-function
             ...
             ...
  ;; Update mode line.
  (force-mode-line-update)))
```

We also normally use two semicolons for comments outside functions.

```
;; This Lisp code is run in Emacs when it is to operate as
;; a server for other processes.
```

If a function has no documentation string, it should instead have a two-semicolon comment right before the function, explaining what the function does and how to call it properly. Explain precisely what each argument means and how the function interprets its possible values. It is much better to convert such comments to documentation strings, though.

‘;;;’

Comments that start with three (or more) semicolons, ‘;;;’, should start at the left margin. We use them for comments that should be considered a heading by Outline minor mode. By default, comments starting with at least three semicolons (followed by a single space and a non-whitespace character) are considered section headings, comments starting with two or fewer are not.

(Historically, triple-semicolon comments have also been used for commenting out lines within a function, but this use is discouraged in favor of using just two semicolons. This also applies when commenting out entire functions; when doing that use two semicolons as well.)

Three semicolons are used for top-level sections, four for sub-sections, five for sub-sub-sections and so on.

Typically libraries have at least four top-level sections. For example when the bodies of all of these sections are hidden:

```
;;; backquote.el --- implement the ` Lisp construct...
;;; Commentary:...
;;; Code:...
;;; backquote.el ends here
```

(In a sense the last line is not a section heading as it must never be followed by any text; after all it marks the end of the file.)

For longer libraries it is advisable to split the code into multiple sections. This can be done by splitting the ‘`Code:`’ section into multiple sub-sections. Even though that was the only recommended approach for a long time, many people have chosen to use multiple top-level code sections instead. You may chose either style.

Using multiple top-level code sections has the advantage that it avoids introducing an additional nesting level but it also means that the section named

‘`Code`’ does not contain all the code, which is awkward. To avoid that, you should put no code at all inside that section; that way it can be considered a separator instead of a section heading.

Finally, we recommend that you don’t end headings with a colon or any other punctuation for that matter. For historic reasons the ‘`Code:`’ and ‘`Commentary:`’ headings end with a colon, but we recommend that you don’t do the same for other headings anyway.

Generally speaking, the *M-;* (`comment-dwim`) command automatically starts a comment of the appropriate type; or indents an existing comment to the right place, depending on the number of semicolons. See Section “Manipulating Comments” in *The GNU Emacs Manual*.

D.8 Conventional Headers for Emacs Libraries

Emacs has conventions for using special comments in Lisp libraries to divide them into sections and give information such as who wrote them. Using a standard format for these items makes it easier for tools (and people) to extract the relevant information. This section explains these conventions, starting with an example:

```
;;; foo.el --- Support for the Foo programming language  -*- lexical-binding: t; -*-

;; Copyright (C) 2010-2021 Your Name

;; Author: Your Name <yourname@example.com>
;; Maintainer: Someone Else <someone@example.com>
;; Created: 14 Jul 2010
;; Keywords: languages
;; URL: https://example.com/foo

;; This file is not part of GNU Emacs.

;; This file is free software...
...
;; along with this file.  If not, see <https://www.gnu.org/licenses/>.
```

The very first line should have this format:

```
;;; filename --- description  -*- lexical-binding: t; -*-
```

The description should be contained in one line. If the file needs to set more variables in the ‘`-*-`’ specification, add it after `lexical-binding`. If this would make the first line too long, use a Local Variables section at the end of the file.

The copyright notice usually lists your name (if you wrote the file). If you have an employer who claims copyright on your work, you might need to list them instead. Do not say that the copyright holder is the Free Software Foundation (or that the file is part of GNU Emacs) unless your file has been accepted into the Emacs distribution. For more information on the form of copyright and license notices, see the guide on the GNU website (`https://www.gnu.org/licenses/gpl-howto.html`).

After the copyright notice come several *header comment* lines, each beginning with ‘`;;` *header-name*`:`’. Here is a table of the conventional possibilities for *header-name*:

‘`Author`’ This header states the name and email address of at least the principal author of the library. If there are multiple authors, list them on continuation lines led

by `;;` and a tab or at least two spaces. We recommend including a contact email address, of the form '`<...>`'. For example:

```
;; Author: Your Name <yourname@example.com>
;;       Someone Else <someone@example.com>
;;       Another Person <another@example.com>
```

'`Maintainer`'
: This header has the same format as the Author header. It lists the person(s) who currently maintain(s) the file (respond to bug reports, etc.).

  If there is no Maintainer header, the person(s) in the Author header is/are presumed to be the maintainer(s). Some files in Emacs use '`emacs-devel@gnu.org`' for the maintainer, which means the author is no longer responsible for the file, and that it is maintained as part of Emacs.

'`Created`'
: This optional line gives the original creation date of the file, and is for historical interest only.

'`Version`'
: If you wish to record version numbers for the individual Lisp program, put them in this line. Lisp files distributed with Emacs generally do not have a '`Version`' header, since the version number of Emacs itself serves the same purpose. If you are distributing a collection of multiple files, we recommend not writing the version in every file, but only the main one.

'`Keywords`'
: This line lists keywords for the `finder-by-keyword` help command. Please use that command to see a list of the meaningful keywords. The command *`M-x checkdoc-package-keywords`* `RET` will find and display any keywords that are not in `finder-known-keywords`. If you set the variable `checkdoc-package-keywords-flag` non-`nil`, checkdoc commands will include the keyword verification in its checks.

  This field is how people will find your package when they're looking for things by topic. To separate the keywords, you can use spaces, commas, or both.

  The name of this field is unfortunate, since people often assume it is the place to write arbitrary keywords that describe their package, rather than just the relevant Finder keywords.

'`URL`'
'`Homepage`'
: These lines state the website of the library.

'`Package-Version`'
: If '`Version`' is not suitable for use by the package manager, then a package can define '`Package-Version`'; it will be used instead. This is handy if '`Version`' is an RCS id or something else that cannot be parsed by `version-to-list`. See Section 43.1 [Packaging Basics], page 1308.

'`Package-Requires`'
: If this exists, it names packages on which the current package depends for proper operation. See Section 43.1 [Packaging Basics], page 1308. This is used by the package manager both at download time (to ensure that a complete set

of packages is downloaded) and at activation time (to ensure that a package is only activated if all its dependencies have been).

Its format is a list of lists on a single line. The `car` of each sub-list is the name of a package, as a symbol. The `cadr` of each sub-list is the minimum acceptable version number, as a string that can be parsed by `version-to-list`. An entry that lacks a version (i.e., an entry which is just a symbol, or a sub-list of one element) is equivalent to entry with version `"0"`. For instance:

```
;; Package-Requires: ((gnus "1.0") (bubbles "2.7.2") cl-lib (seq))
```

Packages that don't need to support Emacs versions older than Emacs 27 can have the '`Package-Requires`' header split across multiple lines, like this:

```
;; Package-Requires: ((emacs "27.1")
;;                    (compat "29.1.4.1"))
```

Note that with this format, you still need to start the list on the same line as '`Package-Requires`'.

The package code automatically defines a package named '`emacs`' with the version number of the currently running Emacs. This can be used to require a minimal version of Emacs for a package.

Just about every Lisp library ought to have the '`Author`' and '`Keywords`' header comment lines. Use the others if they are appropriate. You can also put in header lines with other header names—they have no standard meanings, so they can't do any harm.

We use additional stylized comments to subdivide the contents of the library file. These should be separated from anything else by blank lines. Here is a table of them:

'`;;; Commentary:`'
: This begins introductory comments that explain how the library works. It should come right after the copying permissions, terminated by a '`Change Log`', '`History`' or '`Code`' comment line. This text is used by the Finder package, so it should make sense in that context.

'`;;; Change Log:`'
: This begins an optional log of changes to the file over time. Don't put too much information in this section—it is better to keep the detailed logs in a version control system (as Emacs does) or in a separate `ChangeLog` file. '`History`' is an alternative to '`Change Log`'.

'`;;; Code:`'
: This begins the actual code of the program.

'`;;; `*`filename`*` ends here`'
: This is the *footer line*; it appears at the very end of the file. Its purpose is to enable people to detect truncated versions of the file from the lack of a footer line.

Appendix E GNU Emacs Internals

This chapter describes how the runnable Emacs executable is dumped with the preloaded Lisp libraries in it, how storage is allocated, and some internal aspects of GNU Emacs that may be of interest to C programmers.

E.1 Building Emacs

This section explains the steps involved in building the Emacs executable. You don't have to know this material to build and install Emacs, since the makefiles do all these things automatically. This information is pertinent to Emacs developers.

Building Emacs requires GNU Make version 3.81 or later.

Compilation of the C source files in the `src` directory produces an executable file called `temacs`, also called a *bare impure Emacs*. It contains the Emacs Lisp interpreter and I/O routines, but not the editing commands.

The command `temacs -l loadup` would run `temacs` and direct it to load `loadup.el`. The `loadup` library loads additional Lisp libraries, which set up the normal Emacs editing environment. After this step, the Emacs executable is no longer *bare*.

Because it takes some time to load the standard Lisp files, the `temacs` executable usually isn't run directly by users. Instead, one of the last steps of building Emacs runs the command '`temacs -batch -l loadup --temacs=`*`dump-method`*'. The special option `--temacs` tells `temacs` how to record all the standard preloaded Lisp functions and variables, so that when you subsequently run Emacs, it will start much faster. The `--temacs` option requires an argument *dump-method*, which can be one of the following:

'`pdump`'
: Record the preloaded Lisp data in a *dump file*. This method produces an additional data file which Emacs will load at startup. The produced dump file is usually called `emacs.pdmp`, and is installed in the Emacs `exec-directory` (see Section 25.6 [Help Functions], page 604). This method is the most preferred one, as it does not require Emacs to employ any special techniques of memory allocation, which might get in the way of various memory-layout techniques used by modern systems to enhance security and privacy.

'`pbootstrap`'
: Like '`pdump`', but used while *bootstrapping* Emacs, when no previous Emacs binary and no `*.elc` byte-compiled Lisp files are available. The produced dump file is usually named `bootstrap-emacs.pdmp` in this case.

'`dump`'
: This method causes `temacs` to dump out an executable program, called `emacs`, which has all the standard Lisp files already preloaded into it. (The '`-batch`' argument prevents `temacs` from trying to initialize any of its data on the terminal, so that the tables of terminal information are empty in the dumped Emacs.) This method is also known as *unexec*, because it produces a program file from a running process, and thus is in some sense the opposite of executing a program to start a process. Although this method was the way that Emacs traditionally saved its state, it is now deprecated.

'`bootstrap`'
: Like '`dump`', but used when bootstrapping Emacs with the `unexec` method.

The dumped `emacs` executable (also called a *pure* Emacs) is the one which is installed. If the portable dumper was used to build Emacs, the `emacs` executable is actually an exact copy of `temacs`, and the corresponding `emacs.pdmp` file is installed as well. The variable `preloaded-file-list` stores a list of the preloaded Lisp files recorded in the dump file or in the dumped Emacs executable. If you port Emacs to a new operating system, and are not able to implement dumping of any kind, then Emacs must load `loadup.el` each time it starts.

By default the dumped `emacs` executable records details such as the build time and host name. Use the `--disable-build-details` option of `configure` to suppress these details, so that building and installing Emacs twice from the same sources is more likely to result in identical copies of Emacs.

You can specify additional files to preload by writing a library named `site-load.el` that loads them. You may need to rebuild Emacs with an added definition

```
#define SITELOAD_PURESIZE_EXTRA n
```

to make *n* added bytes of pure space to hold the additional files; see `src/puresize.h`. (Try adding increments of 20000 until it is big enough.) However, the advantage of preloading additional files decreases as machines get faster. On modern machines, it is usually not advisable.

After `loadup.el` reads `site-load.el`, it finds the documentation strings for primitive and preloaded functions (and variables) in the file `etc/DOC` where they are stored, by calling `Snarf-documentation` (see [Accessing Documentation], page 600).

You can specify other Lisp expressions to execute just before dumping by putting them in a library named `site-init.el`. This file is executed after the documentation strings are found.

If you want to preload function or variable definitions, there are three ways you can do this and make their documentation strings accessible when you subsequently run Emacs:

- Arrange to scan these files when producing the `etc/DOC` file, and load them with `site-load.el`.
- Load the files with `site-init.el`, then copy the files into the installation directory for Lisp files when you install Emacs.
- Specify a `nil` value for `byte-compile-dynamic-docstrings` as a local variable in each of these files, and load them with either `site-load.el` or `site-init.el`. (This method has the drawback that the documentation strings take up space in Emacs all the time.)

It is not advisable to put anything in `site-load.el` or `site-init.el` that would alter any of the features that users expect in an ordinary unmodified Emacs. If you feel you must override normal features for your site, do it with `default.el`, so that users can override your changes if they wish. See Section 42.1.1 [Startup Summary], page 1260. Note that if either `site-load.el` or `site-init.el` changes `load-path`, the changes will be lost after dumping. See Section 16.3 [Library Search], page 302. To make a permanent change to `load-path`, use the `--enable-locallisppath` option of `configure`.

In a package that can be preloaded, it is sometimes necessary (or useful) to delay certain evaluations until Emacs subsequently starts up. The vast majority of such cases relate to the values of customizable variables. For example, `tutorial-directory` is a variable defined in `startup.el`, which is preloaded. The default value is set based on `data-directory`. The

variable needs to access the value of `data-directory` when Emacs starts, not when it is dumped, because the Emacs executable has probably been installed in a different location since it was dumped.

`custom-initialize-delay` *symbol value* [Function]
: This function delays the initialization of *symbol* to the next Emacs start. You normally use this function by specifying it as the `:initialize` property of a customizable variable. (The argument *value* is unused, and is provided only for compatibility with the form Custom expects.)

In the unlikely event that you need a more general functionality than `custom-initialize-delay` provides, you can use `before-init-hook` (see Section 42.1.1 [Startup Summary], page 1260).

`dump-emacs-portable` *to-file* **&optional** *track-referrers* [Function]
: This function dumps the current state of Emacs into a dump file *to-file*, using the `pdump` method. Normally, the dump file is called *`emacs-name`*`.dmp`, where *emacs-name* is the name of the Emacs executable file. The optional argument *track-referrers*, if non-`nil`, causes the portable dumper to keep additional information to help track down the provenance of object types that are not yet supported by the `pdump` method.

  Although the portable dumper code can run on many platforms, the dump files that it produces are not portable—they can be loaded only by the Emacs executable that dumped them.

  If you want to use this function in an Emacs that was already dumped, you must run Emacs with the '`-batch`' option.

  If you're including '`.el`' files in the dumped Emacs and that '`.el`' file has code that is normally run at load time, that code won't be run when Emacs starts after dumping. To help work around that problem, you can put functions on the `after-pdump-load-hook` hook. This hook is run when starting Emacs.

`dump-emacs` *to-file from-file* [Function]
: This function dumps the current state of Emacs into an executable file *to-file*, using the `unexec` method. It takes symbols from *from-file* (this is normally the executable file `temacs`).

  This function cannot be used in an Emacs that was already dumped. This function is deprecated, and by default Emacs is built without `unexec` support so this function is not available.

`pdumper-stats` [Function]
: If the current Emacs session restored its state from a dump file, this function returns information about the dump file and the time it took to restore the Emacs state. The value is an alist `((dumped-with-pdumper . t) (load-time . `*`time`*`) (dump-file-name . `*`file`*`))`, where *file* is the name of the dump file, and *time* is the time in seconds it took to restore the state from the dump file. If the current session was not restored from a dump file, the value is `nil`.

E.2 Pure Storage

Emacs Lisp uses two kinds of storage for user-created Lisp objects: *normal storage* and *pure storage*. Normal storage is where all the new data created during an Emacs session are kept (see Section E.3 [Garbage Collection], page 1355). Pure storage is used for certain data in the preloaded standard Lisp files—data that should never change during actual use of Emacs.

Pure storage is allocated only while `temacs` is loading the standard preloaded Lisp libraries. In the file `emacs`, it is marked as read-only (on operating systems that permit this), so that the memory space can be shared by all the Emacs jobs running on the machine at once. Pure storage is not expandable; a fixed amount is allocated when Emacs is compiled, and if that is not sufficient for the preloaded libraries, `temacs` allocates dynamic memory for the part that didn't fit. If Emacs will be dumped using the `pdump` method (see Section E.1 [Building Emacs], page 1351), the pure-space overflow is of no special importance (it just means some of the preloaded stuff cannot be shared with other Emacs jobs). However, if Emacs will be dumped using the now obsolete `unexec` method, the resulting image will work, but garbage collection (see Section E.3 [Garbage Collection], page 1355) is disabled in this situation, causing a memory leak. Such an overflow normally won't happen unless you try to preload additional libraries or add features to the standard ones. Emacs will display a warning about the overflow when it starts, if it was dumped using `unexec`. If this happens, you should increase the compilation parameter `SYSTEM_PURESIZE_EXTRA` in the file `src/puresize.h` and rebuild Emacs.

`purecopy` *object* [Function]

This function makes a copy in pure storage of *object*, and returns it. It copies a string by simply making a new string with the same characters, but without text properties, in pure storage. It recursively copies the contents of vectors and cons cells. It does not make copies of other objects such as symbols, but just returns them unchanged. It signals an error if asked to copy markers.

This function is a no-op except while Emacs is being built and dumped; it is usually called only in preloaded Lisp files.

`pure-bytes-used` [Variable]

The value of this variable is the number of bytes of pure storage allocated so far. Typically, in a dumped Emacs, this number is very close to the total amount of pure storage available—if it were not, we would preallocate less.

`purify-flag` [Variable]

This variable determines whether `defun` should make a copy of the function definition in pure storage. If it is non-`nil`, then the function definition is copied into pure storage.

This flag is `t` while loading all of the basic functions for building Emacs initially (allowing those functions to be shareable and non-collectible). Dumping Emacs as an executable always writes `nil` in this variable, regardless of the value it actually has before and after dumping.

You should not change this flag in a running Emacs.

E.3 Garbage Collection

When a program creates a list or the user defines a new function (such as by loading a library), that data is placed in normal storage. If normal storage runs low, then Emacs asks the operating system to allocate more memory. Different types of Lisp objects, such as symbols, cons cells, small vectors, markers, etc., are segregated in distinct blocks in memory. (Large vectors, long strings, buffers and certain other editing types, which are fairly large, are allocated in individual blocks, one per object; small strings are packed into blocks of 8k bytes, and small vectors are packed into blocks of 4k bytes).

Beyond the basic vector, a lot of objects like markers, overlays and buffers are managed as if they were vectors. The corresponding C data structures include the `union vectorlike_header` field whose `size` member contains the subtype enumerated by `enum pvec_type` and an information about how many `Lisp_Object` fields this structure contains and what the size of the rest data is. This information is needed to calculate the memory footprint of an object, and used by the vector allocation code while iterating over the vector blocks.

It is quite common to use some storage for a while, then release it by (for example) killing a buffer or deleting the last pointer to an object. Emacs provides a *garbage collector* to reclaim this abandoned storage. The garbage collector operates, in essence, by finding and marking all Lisp objects that are still accessible to Lisp programs. To begin with, it assumes all the symbols, their values and associated function definitions, and any data presently on the stack, are accessible. Any objects that can be reached indirectly through other accessible objects are also accessible, but this calculation is done "conservatively", so it may slightly overestimate how many objects that are accessible.

When marking is finished, all objects still unmarked are garbage. No matter what the Lisp program or the user does, it is impossible to refer to them, since there is no longer a way to reach them. Their space might as well be reused, since no one will miss them. The second (sweep) phase of the garbage collector arranges to reuse them. (But since the marking was done "conservatively", not all unused objects are guaranteed to be garbage-collected by any one sweep.)

The sweep phase puts unused cons cells onto a *free list* for future allocation; likewise for symbols and markers. It compacts the accessible strings so they occupy fewer 8k blocks; then it frees the other 8k blocks. Unreachable vectors from vector blocks are coalesced to create largest possible free areas; if a free area spans a complete 4k block, that block is freed. Otherwise, the free area is recorded in a free list array, where each entry corresponds to a free list of areas of the same size. Large vectors, buffers, and other large objects are allocated and freed individually.

> **Common Lisp note:** Unlike other Lisps, GNU Emacs Lisp does not call the garbage collector when the free list is empty. Instead, it simply requests the operating system to allocate more storage, and processing continues until `gc-cons-threshold` bytes have been used.
>
> This means that you can make sure that the garbage collector will not run during a certain portion of a Lisp program by calling the garbage collector explicitly just before it (provided that portion of the program does not use so much space as to force a second garbage collection).

`garbage-collect` [Command]

This command runs a garbage collection, and returns information on the amount of space in use. (Garbage collection can also occur spontaneously if you use more than `gc-cons-threshold` bytes of Lisp data since the previous garbage collection.)

`garbage-collect` returns a list with information on amount of space in use, where each entry has the form '`(`*`name size used`*`)`' or '`(`*`name size used free`*`)`'. In the entry, *name* is a symbol describing the kind of objects this entry represents, *size* is the number of bytes used by each one, *used* is the number of those objects that were found live in the heap, and optional *free* is the number of those objects that are not live but that Emacs keeps around for future allocations. So an overall result is:

```
((conses cons-size used-conses free-conses)
 (symbols symbol-size used-symbols free-symbols)
 (strings string-size used-strings free-strings)
 (string-bytes byte-size used-bytes)
 (vectors vector-size used-vectors)
 (vector-slots slot-size used-slots free-slots)
 (floats float-size used-floats free-floats)
 (intervals interval-size used-intervals free-intervals)
 (buffers buffer-size used-buffers)
 (heap unit-size total-size free-size))
```

Here is an example:

```
(garbage-collect)
      ⇒ ((conses 16 49126 8058) (symbols 48 14607 0)
                (strings 32 2942 2607)
                (string-bytes 1 78607) (vectors 16 7247)
                (vector-slots 8 341609 29474) (floats 8 71 102)
                (intervals 56 27 26) (buffers 944 8)
                (heap 1024 11715 2678))
```

Below is a table explaining each element. Note that last `heap` entry is optional and present only if an underlying `malloc` implementation provides `mallinfo` function.

cons-size Internal size of a cons cell, i.e., `sizeof (struct Lisp_Cons)`.

used-conses
: The number of cons cells in use.

free-conses
: The number of cons cells for which space has been obtained from the operating system, but that are not currently being used.

symbol-size
: Internal size of a symbol, i.e., `sizeof (struct Lisp_Symbol)`.

used-symbols
: The number of symbols in use.

free-symbols
: The number of symbols for which space has been obtained from the operating system, but that are not currently being used.

string-size Internal size of a string header, i.e., `sizeof (struct Lisp_String)`.

used-strings
The number of string headers in use.

free-strings
The number of string headers for which space has been obtained from the operating system, but that are not currently being used.

byte-size This is used for convenience and equals to `sizeof (char)`.

used-bytes The total size of all string data in bytes.

vector-size Size in bytes of a vector of length 1, including its header.

used-vectors
The number of vector headers allocated from the vector blocks.

slot-size Internal size of a vector slot, always equal to `sizeof (Lisp_Object)`.

used-slots The number of slots in all used vectors. Slot counts might include some or all overhead from vector headers, depending on the platform.

free-slots The number of free slots in all vector blocks.

float-size Internal size of a float object, i.e., `sizeof (struct Lisp_Float)`. (Do not confuse it with the native platform `float` or `double`.)

used-floats
The number of floats in use.

free-floats The number of floats for which space has been obtained from the operating system, but that are not currently being used.

interval-size
Internal size of an interval object, i.e., `sizeof (struct interval)`.

used-intervals
The number of intervals in use.

free-intervals
The number of intervals for which space has been obtained from the operating system, but that are not currently being used.

buffer-size Internal size of a buffer, i.e., `sizeof (struct buffer)`. (Do not confuse with the value returned by `buffer-size` function.)

used-buffers
The number of buffer objects in use. This includes killed buffers invisible to users, i.e., all buffers in `all_buffers` list.

unit-size The unit of heap space measurement, always equal to 1024 bytes.

total-size Total heap size, in *unit-size* units.

free-size Heap space which is not currently used, in *unit-size* units.

If there was overflow in pure space (see Section E.2 [Pure Storage], page 1354), and Emacs was dumped using the (now obsolete) `unexec` method (see Section E.1 [Building Emacs], page 1351), then `garbage-collect` returns `nil`, because a real garbage collection cannot be done in that case.

`garbage-collection-messages` [User Option]
: If this variable is non-`nil`, Emacs displays a message at the beginning and end of garbage collection. The default value is `nil`.

`post-gc-hook` [Variable]
: This is a normal hook that is run at the end of garbage collection. Garbage collection is inhibited while the hook functions run, so be careful writing them.

`gc-cons-threshold` [User Option]
: The value of this variable is the number of bytes of storage that must be allocated for Lisp objects after one garbage collection in order to trigger another garbage collection. You can use the result returned by `garbage-collect` to get an information about size of the particular object type; space allocated to the contents of buffers does not count.

  The initial threshold value is `GC_DEFAULT_THRESHOLD`, defined in `alloc.c`. Since it's defined in `word_size` units, the value is 400,000 for the default 32-bit configuration and 800,000 for the 64-bit one. If you specify a larger value, garbage collection will happen less often. This reduces the amount of time spent garbage collecting (so Lisp programs will run faster between cycles of garbage collection that happen more rarely), but increases total memory use. You may want to do this when running a program that creates lots of Lisp data, especially if you need it to run faster. However, we recommend against increasing the threshold for prolonged periods of time, and advise that you never set it higher than needed for the program to run in reasonable time. Using thresholds higher than necessary could potentially cause higher system-wide memory pressure, and also make each garbage-collection cycle take much more time, and should therefore be avoided.

  You can make collections more frequent by specifying a smaller value, down to 1/10th of `GC_DEFAULT_THRESHOLD`. A value less than this minimum will remain in effect only until the subsequent garbage collection, at which time `garbage-collect` will set the threshold back to the minimum.

`gc-cons-percentage` [User Option]
: The value of this variable specifies the amount of consing before a garbage collection occurs, as a fraction of the current heap size. This criterion and `gc-cons-threshold` apply in parallel, and garbage collection occurs only when both criteria are satisfied.

  As the heap size increases, the time to perform a garbage collection increases. Thus, it can be desirable to do them less frequently in proportion.

  As with `gc-cons-threshold`, do not enlarge this more than necessary, and never for prolonged periods of time.

Control over the garbage collector via `gc-cons-threshold` and `gc-cons-percentage` is only approximate. Although Emacs checks for threshold exhaustion regularly, for efficiency reasons it does not do so immediately after every change to the heap or to `gc-cons-threshold` or `gc-cons-percentage`, so exhausting the threshold does not immediately trigger garbage collection. Also, for efficiency in threshold calculations Emacs approximates the heap size, which counts the bytes used by currently-accessible objects in the heap.

The value returned by `garbage-collect` describes the amount of memory used by Lisp data, broken down by data type. By contrast, the function `memory-limit` provides information on the total amount of memory Emacs is currently using.

`memory-limit` [Function]
: This function returns an estimate of the total amount of bytes of virtual memory that Emacs is currently using, divided by 1024. You can use this to get a general idea of how your actions affect the memory usage.

`memory-full` [Variable]
: This variable is `t` if Emacs is nearly out of memory for Lisp objects, and `nil` otherwise.

`memory-use-counts` [Function]
: This returns a list of numbers that count the number of objects created in this Emacs session. Each of these counters increments for a certain kind of object. See the documentation string for details.

`memory-info` [Function]
: This functions returns an amount of total system memory and how much of it is free. On an unsupported system, the value may be `nil`.

  If `default-directory` points to a remote host, memory information of that host is returned.

`gcs-done` [Variable]
: This variable contains the total number of garbage collections done so far in this Emacs session.

`gc-elapsed` [Variable]
: This variable contains the total number of seconds of elapsed time during garbage collection so far in this Emacs session, as a floating-point number.

`memory-report` [Function]
: It can sometimes be useful to see where Emacs is using memory (in various variables, buffers, and caches). This command will open a new buffer (called ‘`"*Memory Report*"`’) that will give an overview, in addition to listing the “largest” buffers and variables.

  All the data here is approximate, because there's really no consistent way to compute the size of a variable. For instance, two variables may share parts of a data structure, and this will be counted twice, but this command may still give a useful high-level overview of which parts of Emacs are using memory.

E.4 Stack-allocated Objects

The garbage collector described above is used to manage data visible from Lisp programs, as well as most of the data internally used by the Lisp interpreter. Sometimes it may be useful to allocate temporary internal objects using the C stack of the interpreter. This can help performance, as stack allocation is typically faster than using heap memory to allocate and the garbage collector to free. The downside is that using such objects after they are freed results in undefined behavior, so uses should be well thought out and carefully debugged

by using the `GC_CHECK_MARKED_OBJECTS` feature (see `src/alloc.c`). In particular, stack-allocated objects should never be made visible to user Lisp code.

Currently, cons cells and strings can be allocated this way. This is implemented by C macros like `AUTO_CONS` and `AUTO_STRING` that define a named `Lisp_Object` with block lifetime. These objects are not freed by the garbage collector; instead, they have automatic storage duration, i.e., they are allocated like local variables and are automatically freed at the end of execution of the C block that defined the object.

For performance reasons, stack-allocated strings are limited to ASCII characters, and many of these strings are immutable, i.e., calling `ASET` on them produces undefined behavior.

E.5 Memory Usage

These functions and variables give information about the total amount of memory allocation that Emacs has done, broken down by data type. Note the difference between these and the values returned by `garbage-collect`; those count objects that currently exist, but these count the number or size of all allocations, including those for objects that have since been freed.

`cons-cells-consed` [Variable]
: The total number of cons cells that have been allocated so far in this Emacs session.

`floats-consed` [Variable]
: The total number of floats that have been allocated so far in this Emacs session.

`vector-cells-consed` [Variable]
: The total number of vector cells that have been allocated so far in this Emacs session. This includes vector-like objects such as markers and overlays, plus certain objects not visible to users.

`symbols-consed` [Variable]
: The total number of symbols that have been allocated so far in this Emacs session.

`string-chars-consed` [Variable]
: The total number of string characters that have been allocated so far in this session.

`intervals-consed` [Variable]
: The total number of intervals that have been allocated so far in this Emacs session.

`strings-consed` [Variable]
: The total number of strings that have been allocated so far in this Emacs session.

E.6 C Dialect

The C part of Emacs is portable to C99 or later: C11-specific features such as '`<stdalign.h>`' and '`_Noreturn`' are not used without a check, typically at configuration time, and the Emacs build procedure provides a substitute implementation if necessary. Some C11 features, such as anonymous structures and unions, are too difficult to emulate, so they are avoided entirely.

At some point in the future the base C dialect will no doubt change to C11.

E.7 Writing Emacs Primitives

Lisp primitives are Lisp functions implemented in C. The details of interfacing the C function so that Lisp can call it are handled by a few C macros. The only way to really understand how to write new C code is to read the source, but we can explain some things here.

An example of a special form is the definition of `or`, from `eval.c`. (An ordinary function would have the same general appearance.)

```
DEFUN ("or", For, Sor, 0, UNEVALLED, 0,
       doc: /* Eval args until one of them yields non-nil,
then return that value.
The remaining args are not evalled at all.
If all args return nil, return nil.
usage: (or CONDITIONS...)  */)
  (Lisp_Object args)
{
  Lisp_Object val = Qnil;

  while (CONSP (args))
    {
      val = eval_sub (XCAR (args));
      if (!NILP (val))
        break;
      args = XCDR (args);
      maybe_quit ();
    }

  return val;
}
```

Let's start with a precise explanation of the arguments to the `DEFUN` macro. Here is a template for them:

```
DEFUN (lname, fname, sname, min, max, interactive, doc)
```

lname This is the name of the Lisp symbol to define as the function name; in the example above, it is `or`.

fname This is the C function name for this function. This is the name that is used in C code for calling the function. The name is, by convention, '`F`' prepended to the Lisp name, with all dashes ('`-`') in the Lisp name changed to underscores. Thus, to call this function from C code, call `For`.

sname This is a C variable name to use for a structure that holds the data for the subr object that represents the function in Lisp. This structure conveys the Lisp symbol name to the initialization routine that will create the symbol and store the subr object as its definition. By convention, this name is always *fname* with '`F`' replaced with '`S`'.

min This is the minimum number of arguments that the function requires. The function `or` allows a minimum of zero arguments.

max This is the maximum number of arguments that the function accepts, if there is a fixed maximum. Alternatively, it can be `UNEVALLED`, indicating a special form that receives unevaluated arguments, or `MANY`, indicating an unlimited number

of evaluated arguments (the equivalent of `&rest`). Both `UNEVALLED` and `MANY` are macros. If *max* is a number, it must be more than *min* but less than 8.

interactive

This is an interactive specification, a string such as might be used as the argument of `interactive` in a Lisp function (see Section 22.2.1 [Using Interactive], page 425). In the case of `or`, it is `0` (a null pointer), indicating that `or` cannot be called interactively. A value of `""` indicates a function that should receive no arguments when called interactively. If the value begins with a '`"(`', the string is evaluated as a Lisp form. For example:

```
DEFUN ("foo", Ffoo, Sfoo, 0, 3,
       "(list (read-char-by-name \"Insert character: \")\
              (prefix-numeric-value current-prefix-arg)\
              t)",
       doc: /* ... */)
```

doc This is the documentation string. It uses C comment syntax rather than C string syntax because comment syntax requires nothing special to include multiple lines. The '`doc:`' identifies the comment that follows as the documentation string. The '`/*`' and '`*/`' delimiters that begin and end the comment are not part of the documentation string.

If the last line of the documentation string begins with the keyword '`usage:`', the rest of the line is treated as the argument list for documentation purposes. This way, you can use different argument names in the documentation string from the ones used in the C code. '`usage:`' is required if the function has an unlimited number of arguments.

Some primitives have multiple definitions, one per platform (e.g., `x-create-frame`). In such cases, rather than writing the same documentation string in each definition, only one definition has the actual documentation. The others have placeholders beginning with '`SKIP`', which are ignored by the function that parses the `DOC` file.

All the usual rules for documentation strings in Lisp code (see Section D.6 [Documentation Tips], page 1342) apply to C code documentation strings too.

The documentation string can be followed by a list of C function attributes for the C function that implements the primitive, like this:

```
DEFUN ("bar", Fbar, Sbar, 0, UNEVALLED, 0
       doc: /* ... */
       attributes: attr1 attr2 ...)
```

You can specify more than a single attribute, one after the other. Currently, only the following attributes are recognized:

`noreturn` Declares the C function as one that never returns. This corresponds to the C11 keyword `_Noreturn` and to `__attribute__ ((__noreturn__))` attribute of GCC (see Section "Function Attributes" in *Using the GNU Compiler Collection*).

`const` Declares that the function does not examine any values except its arguments, and has no effects except the return value. This corresponds to `__attribute__ ((__const__))` attribute of GCC.

`noinline` This corresponds to `__attribute__ ((__noinline__))` attribute of GCC, which prevents the function from being considered for inlining. This might be needed, e.g., to countermand effects of link-time optimizations on stack-based variables.

After the call to the `DEFUN` macro, you must write the argument list for the C function, including the types for the arguments. If the primitive accepts a fixed maximum number of Lisp arguments, there must be one C argument for each Lisp argument, and each argument must be of type `Lisp_Object`. (Various macros and functions for creating values of type `Lisp_Object` are declared in the file `lisp.h`.) If the primitive is a special form, it must accept a Lisp list containing its unevaluated Lisp arguments as a single argument of type `Lisp_Object`. If the primitive has no upper limit on the number of evaluated Lisp arguments, it must have exactly two C arguments: the first is the number of Lisp arguments, and the second is the address of a block containing their values. These have types `ptrdiff_t` and `Lisp_Object *`, respectively. Since `Lisp_Object` can hold any Lisp object of any data type, you can determine the actual data type only at run time; so if you want a primitive to accept only a certain type of argument, you must check the type explicitly using a suitable predicate (see Section 2.7 [Type Predicates], page 31).

Within the function `For` itself, the local variable `args` refers to objects controlled by Emacs's stack-marking garbage collector. Although the garbage collector does not reclaim objects reachable from C `Lisp_Object` stack variables, it may move some of the components of an object, such as the contents of a string or the text of a buffer. Therefore, functions that access these components must take care to refetch their addresses after performing Lisp evaluation. This means that instead of keeping C pointers to string contents or buffer text, the code should keep the buffer or string position, and recompute the C pointer from the position after performing Lisp evaluation. Lisp evaluation can occur via calls to `eval_sub` or `Feval`, either directly or indirectly.

Note the call to `maybe_quit` inside the loop: this function checks whether the user pressed *`C-g`*, and if so, aborts the processing. You should do that in any loop that can potentially require a large number of iterations; in this case, the list of arguments could be very long. This increases Emacs responsiveness and improves user experience.

You must not use C initializers for static or global variables unless the variables are never written once Emacs is dumped. These variables with initializers are allocated in an area of memory that becomes read-only (on certain operating systems) as a result of dumping Emacs. See Section E.2 [Pure Storage], page 1354.

Defining the C function is not enough to make a Lisp primitive available; you must also create the Lisp symbol for the primitive and store a suitable subr object in its function cell. The code looks like this:

```
defsubr (&sname);
```

Here *sname* is the name you used as the third argument to `DEFUN`.

If you add a new primitive to a file that already has Lisp primitives defined in it, find the function (near the end of the file) named `syms_of_`*`something`*, and add the call to

`defsubr` there. If the file doesn't have this function, or if you create a new file, add to it a `syms_of_`*`filename`* (e.g., `syms_of_myfile`). Then find the spot in `emacs.c` where all of these functions are called, and add a call to `syms_of_`*`filename`* there.

The function `syms_of_`*`filename`* is also the place to define any C variables that are to be visible as Lisp variables. `DEFVAR_LISP` makes a C variable of type `Lisp_Object` visible in Lisp. `DEFVAR_INT` makes a C variable of type `int` visible in Lisp with a value that is always an integer. `DEFVAR_BOOL` makes a C variable of type `int` visible in Lisp with a value that is either `t` or `nil`. Note that variables defined with `DEFVAR_BOOL` are automatically added to the list `byte-boolean-vars` used by the byte compiler.

These macros all expect three arguments:

`lname` The name of the variable to be used by Lisp programs.

`vname` The name of the variable in the C sources.

`doc` The documentation for the variable, as a C comment. See Section 25.1 [Documentation Basics], page 597, for more details.

By convention, when defining variables of a "native" type (`int` and `bool`), the name of the C variable is the name of the Lisp variable with `-` replaced by `_`. When the variable has type `Lisp_Object`, the convention is to also prefix the C variable name with `V`. i.e.

```
DEFVAR_INT ("my-int-variable", my_int_variable,
            doc: /* An integer variable.  */);

DEFVAR_LISP ("my-lisp-variable", Vmy_lisp_variable,
            doc: /* A Lisp variable.  */);
```

There are situations in Lisp where you need to refer to the symbol itself rather than the value of that symbol. One such case is when temporarily overriding the value of a variable, which in Lisp is done with `let`. In C sources, this is done by defining a corresponding, constant symbol, and using `specbind`. By convention, `Qmy_lisp_variable` corresponds to `Vmy_lisp_variable`; to define it, use the `DEFSYM` macro. i.e.

```
DEFSYM (Qmy_lisp_variable, "my-lisp-variable");
```

To perform the actual binding:

```
specbind (Qmy_lisp_variable, Qt);
```

In Lisp symbols sometimes need to be quoted, to achieve the same effect in C you again use the corresponding constant symbol `Qmy_lisp_variable`. For example, when creating a buffer-local variable (see Section 12.11 [Buffer-Local Variables], page 207) in Lisp you would write:

```
(make-variable-buffer-local 'my-lisp-variable)
```

In C the corresponding code uses `Fmake_variable_buffer_local` in combination with `DEFSYM`, i.e.

```
DEFSYM (Qmy_lisp_variable, "my-lisp-variable");
Fmake_variable_buffer_local (Qmy_lisp_variable);
```

If you want to make a Lisp variable that is defined in C behave like one declared with `defcustom`, add an appropriate entry to `cus-start.el`. See Section 15.3 [Variable Definitions], page 281, for a description of the format to use.

If you directly define a file-scope C variable of type `Lisp_Object`, you must protect it from garbage-collection by calling `staticpro` in `syms_of_`*`filename`*, like this:

```
staticpro (&variable);
```

Here is another example function, with more complicated arguments. This comes from the code in `window.c`, and it demonstrates the use of macros and functions to manipulate Lisp objects.

```
DEFUN ("coordinates-in-window-p", Fcoordinates_in_window_p,
       Scoordinates_in_window_p, 2, 2, 0,
       doc: /* Return non-nil if COORDINATES are in WINDOW.
  ...
  or `right-margin' is returned.  */)
  (register Lisp_Object coordinates, Lisp_Object window)
{
  struct window *w;
  struct frame *f;
  int x, y;
  Lisp_Object lx, ly;

  w = decode_live_window (window);
  f = XFRAME (w->frame);
  CHECK_CONS (coordinates);
  lx = Fcar (coordinates);
  ly = Fcdr (coordinates);
  CHECK_NUMBER (lx);
  CHECK_NUMBER (ly);
  x = FRAME_PIXEL_X_FROM_CANON_X (f, lx) + FRAME_INTERNAL_BORDER_WIDTH (f);
  y = FRAME_PIXEL_Y_FROM_CANON_Y (f, ly) + FRAME_INTERNAL_BORDER_WIDTH (f);

  switch (coordinates_in_window (w, x, y))
    {
    case ON_NOTHING:            /* NOT in window at all.  */
      return Qnil;

    ...

    case ON_MODE_LINE:          /* In mode line of window.  */
      return Qmode_line;

    ...

    case ON_SCROLL_BAR:         /* On scroll-bar of window.  */
      /* Historically we are supposed to return nil in this case.  */
      return Qnil;

    default:
      emacs_abort ();
    }
}
```

Note that C code cannot call functions by name unless they are defined in C. The way to call a function written in Lisp is to use `Ffuncall`, which embodies the Lisp function `funcall`. Since the Lisp function `funcall` accepts an unlimited number of arguments, in C it takes two: the number of Lisp-level arguments, and a one-dimensional array containing their values. The first Lisp-level argument is the Lisp function to call, and the rest are the arguments to pass to it.

The C functions `call0`, `call1`, `call2`, and so on, provide handy ways to call a Lisp function conveniently with a fixed number of arguments. They work by calling `Ffuncall`.

`eval.c` is a very good file to look through for examples; `lisp.h` contains the definitions for some important macros and functions.

If you define a function which is side-effect free or pure, give it a non-`nil` `side-effect-free` or `pure` property, respectively (see Section 9.4.2 [Standard Properties], page 139).

E.8 Writing Dynamically-Loaded Modules

This section describes the Emacs module API and how to use it as part of writing extension modules for Emacs. The module API is defined in the C programming language, therefore the description and the examples in this section assume the module is written in C. For other programming languages, you will need to use the appropriate bindings, interfaces and facilities for calling C code. Emacs C code requires a C99 or later compiler (see Section E.6 [C Dialect], page 1360), and so the code examples in this section also follow that standard.

Writing a module and integrating it into Emacs comprises the following tasks:

- Writing initialization code for the module.
- Writing one or more module functions.
- Communicating values and objects between Emacs and your module functions.
- Handling of error conditions and nonlocal exits.

The following subsections describe these tasks and the API itself in more detail.

Once your module is written, compile it to produce a shared library, according to the conventions of the underlying platform. Then place the shared library in a directory mentioned in `load-path` (see Section 16.3 [Library Search], page 302), where Emacs will find it.

If you wish to verify the conformance of a module to the Emacs dynamic module API, invoke Emacs with the *`--module-assertions`* option. See Section "Initial Options" in *The GNU Emacs Manual*.

E.8.1 Module Initialization Code

Begin your module by including the header file `emacs-module.h` and defining the GPL compatibility symbol:

```
#include <emacs-module.h>

int plugin_is_GPL_compatible;
```

The `emacs-module.h` file is installed into your system's include tree as part of the Emacs installation. Alternatively, you can find it in the Emacs source tree.

Next, write an initialization function for the module.

`int emacs_module_init (struct emacs_runtime *`*`runtime`*`)` [Function]

Emacs calls this function when it loads a module. If a module does not export a function named `emacs_module_init`, trying to load the module will signal an error. The initialization function should return zero if the initialization succeeds, non-zero otherwise. In the latter case, Emacs will signal an error, and the loading of the module will fail. If the user presses *`C-g`* during the initialization, Emacs ignores the return value of the initialization function and quits (see Section 22.11 [Quitting],

page 472). (If needed, you can catch user quitting inside the initialization function, see [should_quit], page 1379.)

The argument *runtime* is a pointer to a C `struct` that includes 2 public fields: `size`, which provides the size of the structure in bytes; and `get_environment`, which provides a pointer to a function that allows the module initialization function access to the Emacs environment object and its interfaces.

The initialization function should perform whatever initialization is required for the module. In addition, it can perform the following tasks:

Compatibility verification

A module can verify that the Emacs executable which loads the module is compatible with the module, by comparing the `size` member of the *runtime* structure with the value compiled into the module:

```
int
emacs_module_init (struct emacs_runtime *runtime)
{
  if (runtime->size < sizeof (*runtime))
    return 1;
}
```

If the size of the runtime object passed to the module is smaller than what it expects, it means the module was compiled for an Emacs version newer (later) than the one which attempts to load it, i.e. the module might be incompatible with the Emacs binary.

In addition, a module can verify the compatibility of the module API with what the module expects. The following sample code assumes it is part of the `emacs_module_init` function shown above:

```
emacs_env *env = runtime->get_environment (runtime);
if (env->size < sizeof (*env))
  return 2;
```

This calls the `get_environment` function using the pointer provided in the `runtime` structure to retrieve a pointer to the API's *environment*, a C `struct` which also has a `size` field holding the size of the structure in bytes.

Finally, you can write a module that will work with older versions of Emacs, by comparing the size of the environment passed by Emacs with known sizes, like this:

```
emacs_env *env = runtime->get_environment (runtime);
if (env->size >= sizeof (struct emacs_env_26))
  emacs_version = 26;  /* Emacs 26 or later.  */
else if (env->size >= sizeof (struct emacs_env_25))
  emacs_version = 25;
else
  return 2; /* Unknown or unsupported version.  */
```

This works because later Emacs versions always *add* members to the environment, never *remove* any members, so the size can only grow with

new Emacs releases. Given the version of Emacs, the module can use only the parts of the module API that existed in that version, since those parts are identical in later versions.

`emacs-module.h` defines a preprocessor macro `EMACS_MAJOR_VERSION`. It expands to an integer literal which is the latest major version of Emacs supported by the header. See Section 1.4 [Version Info], page 6. Note that the value of `EMACS_MAJOR_VERSION` is a compile-time constant and does not represent the version of Emacs that is currently running and has loaded your module. If you want your module to be compatible with various versions of `emacs-module.h` as well as various versions of Emacs, you can use conditional compilation based on `EMACS_MAJOR_VERSION`.

We recommend that modules always perform the compatibility verification, unless they do their job entirely in the initialization function, and don't access any Lisp objects or use any Emacs functions accessible through the environment structure.

Binding module functions to Lisp symbols
: This gives the module functions names so that Lisp code could call it by that name. We describe how to do this in Section E.8.2 [Module Functions], page 1368, below.

E.8.2 Writing Module Functions

The main reason for writing an Emacs module is to make additional functions available to Lisp programs that load the module. This subsection describes how to write such *module functions*.

A module function has the following general form and signature:

`emacs_value emacs_function (emacs_env *env, ptrdiff_t nargs, emacs_value *args, void *data)` [Function]

The *env* argument provides a pointer to the API environment, needed to access Emacs objects and functions. The *nargs* argument is the required number of arguments, which can be zero (see `make_function` below for more flexible specification of the argument number), and *args* is a pointer to the array of the function arguments. The argument *data* points to additional data required by the function, which was arranged when `make_function` (see below) was called to create an Emacs function from `emacs_function`.

Module functions use the type `emacs_value` to communicate Lisp objects between Emacs and the module (see Section E.8.3 [Module Values], page 1371). The API, described below and in the following subsections, provides facilities for conversion between basic C data types and the corresponding `emacs_value` objects.

In the module function's body, do *not* attempt to access elements of the *args* array beyond the index *nargs*`-1`: memory for the *args* array is allocated exactly to accommodate *nargs* values, and accessing beyond that will most probably crash your module. In particular, if the value of *nargs* passed to the function at run time is zero, it must not access *args* at all, as no memory will have been allocated for it in that case.

A module function always returns a value. If the function returns normally, the Lisp code which called it will see the Lisp object corresponding to the `emacs_value` value the function returned. However, if the user typed *C-g*, or if the module function or its callees signaled an error or exited nonlocally (see Section E.8.5 [Module Nonlocal], page 1380), Emacs will ignore the returned value and quit or throw as it does when Lisp code encounters the same situations.

The header `emacs-module.h` provides the type `emacs_function` as an alias type for a function pointer to a module function.

After writing your C code for a module function, you should make a Lisp function object from it using the `make_function` function, whose pointer is provided in the environment (recall that the pointer to the environment is returned by `get_environment`). This is normally done in the module initialization function (see [module initialization function], page 1366), after verifying the API compatibility.

`emacs_value make_function (emacs_env *`*`env`*`, ptrdiff_t` *`min_arity`*`, ptrdiff_t` *`max_arity`*`, emacs_function` *`func`*`, const char *`*`docstring`*`, void *`*`data`*`)` [Function]

This returns an Emacs function created from the C function *func*, whose signature is as described for `emacs_function` above. The arguments *min_arity* and *max_arity* specify the minimum and maximum number of arguments that *func* can accept. The *max_arity* argument can have the special value `emacs_variadic_function`, which makes the function accept an unlimited number of arguments, like the `&rest` keyword in Lisp (see Section 13.2.3 [Argument List], page 234).

The argument *data* is a way to arrange for arbitrary additional data to be passed to *func* when it is called. Whatever pointer is passed to `make_function` will be passed unaltered to *func*.

The argument *docstring* specifies the documentation string for the function. It should be either an ASCII string, or a UTF-8 encoded non-ASCII string, or a `NULL` pointer; in the latter case the function will have no documentation. The documentation string can end with a line that specifies the advertised calling convention, see Section 13.2.4 [Function Documentation], page 235.

Since every module function must accept the pointer to the environment as its first argument, the call to `make_function` could be made from any module function, but you will normally want to do that from the module initialization function, so that all the module functions are known to Emacs once the module is loaded.

Finally, you should bind the Lisp function to a symbol, so that Lisp code could call your function by name. For that, use the module API function `intern` (see [intern], page 1379) whose pointer is also provided in the environment that module functions can access.

Combining the above steps, code that arranges for a C function `module_func` to be callable as `module-func` from Lisp will look like this, as part of the module initialization function:

```
emacs_env *env = runtime->get_environment (runtime);
emacs_value func = env->make_function (env, min_arity, max_arity,
                                       module_func, docstring, data);
```

```
emacs_value symbol = env->intern (env, "module-func");
emacs_value args[] = {symbol, func};
env->funcall (env, env->intern (env, "defalias"), 2, args);
```

This makes the symbol `module-func` known to Emacs by calling `env->intern`, then invokes `defalias` from Emacs to bind the function to that symbol. Note that it is possible to use `fset` instead of `defalias`; the differences are described in Section 13.4 [Defining Functions], page 237.

Module functions including the `emacs_module_init` function (see [module initialization function], page 1366) may only interact with Emacs by calling environment functions from some live `emacs_env` pointer while being called directly or indirectly from Emacs. In other words, if a module function wants to call Lisp functions or Emacs primitives, convert `emacs_value` objects to and from C datatypes (see Section E.8.3 [Module Values], page 1371), or interact with Emacs in any other way, some call from Emacs to `emacs_module_init` or to a module function must be in the call stack. Module functions may not interact with Emacs while garbage collection is running; see Section E.3 [Garbage Collection], page 1355. They may only interact with Emacs from Lisp interpreter threads (including the main thread) created by Emacs; see Chapter 39 [Threads], page 1076. The *`--module-assertions`* command-line option can detect some violations of the above requirements. See Section "Initial Options" in *The GNU Emacs Manual*.

Using the module API, it is possible to define more complex function and data types: inline functions, macros, etc. However, the resulting C code will be cumbersome and hard to read. Therefore, we recommend that you limit the module code which creates functions and data structures to the absolute minimum, and leave the rest for a Lisp package that will accompany your module, because doing these additional tasks in Lisp is much easier, and will produce a much more readable code. For example, given a module function `module-func` defined as above, one way of making a macro `module-macro` based on it is with the following simple Lisp wrapper:

```
(defmacro module-macro (&rest args)
  "Documentation string for the macro."
  (module-func args))
```

The Lisp package which goes with your module could then load the module using the `load` primitive (see Section 16.11 [Dynamic Modules], page 315) when the package is loaded into Emacs.

By default, module functions created by `make_function` are not interactive. To make them interactive, you can use the following function.

`void make_interactive (emacs_env *`*`env`*`, emacs_value` *`function`*`, emacs_value` *`spec`*`)` [Function]

This function, which is available since Emacs 28, makes the function *function* interactive using the interactive specification *spec*. Emacs interprets *spec* like the argument to the `interactive` form. Section 22.2.1 [Using Interactive], page 425, and see Section 22.2.2 [Interactive Codes], page 428. *function* must be an Emacs module function returned by `make_function`.

Note that there is no native module support for retrieving the interactive specification of a module function. Use the function `interactive-form` for that. Section 22.2.1 [Using

Interactive], page 425. It is not possible to make a module function non-interactive once you have made it interactive using `make_interactive`.

If you want to run some code when a module function object (i.e., an object returned by `make_function`) is garbage-collected, you can install a *function finalizer*. Function finalizers are available since Emacs 28. For example, if you have passed some heap-allocated structure to the *data* argument of `make_function`, you can use the finalizer to deallocate the structure. See Section "Basic Allocation" in `libc`, and see Section "Freeing after Malloc" in `libc`. The finalizer function has the following signature:

```
void finalizer (void *data)
```

Here, *data* receives the value passed to *data* when calling `make_function`. Note that the finalizer can't interact with Emacs in any way.

Directly after calling `make_function`, the newly-created function doesn't have a finalizer. Use `set_function_finalizer` to add one, if desired.

`void emacs_finalizer (void *ptr)` [Function]
: The header `emacs-module.h` provides the type `emacs_finalizer` as a type alias for an Emacs finalizer function.

`emacs_finalizer get_function_finalizer (emacs_env *env, emacs_value arg)` [Function]
: This function, which is available since Emacs 28, returns the function finalizer associated with the module function represented by *arg*. *arg* must refer to a module function, that is, an object returned by `make_function`. If no finalizer is associated with the function, `NULL` is returned.

`void set_function_finalizer (emacs_env *env, emacs_value arg, emacs_finalizer fin)` [Function]
: This function, which is available since Emacs 28, sets the function finalizer associated with the module function represented by *arg* to *fin*. *arg* must refer to a module function, that is, an object returned by `make_function`. *fin* can either be `NULL` to clear *arg*'s function finalizer, or a pointer to a function to be called when the object represented by *arg* is garbage-collected. At most one function finalizer can be set per function; if *arg* already has a finalizer, it is replaced by *fin*.

E.8.3 Conversion Between Lisp and Module Values

With very few exceptions, most modules need to exchange data with Lisp programs that call them: accept arguments to module functions and return values from module functions. For this purpose, the module API provides the `emacs_value` type, which represents Emacs Lisp objects communicated via the API; it is the functional equivalent of the `Lisp_Object` type used in Emacs C primitives (see Section E.7 [Writing Emacs Primitives], page 1361). This section describes the parts of the module API that allow to create `emacs_value` objects corresponding to basic Lisp data types, and how to access from C data in `emacs_value` objects that correspond to Lisp objects.

All of the functions described below are actually *function pointers* provided via the pointer to the environment which every module function accepts. Therefore, module code should call these functions through the environment pointer, like this:

```
emacs_env *env;  /* the environment pointer */
```

```
env->some_function (arguments...);
```

The `emacs_env` pointer will usually come from the first argument to the module function, or from the call to `get_environment` if you need the environment in the module initialization function.

Most of the functions described below became available in Emacs 25, the first Emacs release that supported dynamic modules. For the few functions that became available in later Emacs releases, we mention the first Emacs version that supported them.

The following API functions extract values of various C data types from `emacs_value` objects. They all raise the `wrong-type-argument` error condition (see Section 2.7 [Type Predicates], page 31) if the argument `emacs_value` object is not of the type expected by the function. See Section E.8.5 [Module Nonlocal], page 1380, for details of how signaling errors works in Emacs modules, and how to catch error conditions inside the module before they are reported to Emacs. The API function `type_of` (see Section E.8.4 [Module Misc], page 1378) can be used to obtain the type of a `emacs_value` object.

`intmax_t extract_integer (emacs_env *env, emacs_value arg)` [Function]
: This function returns the value of a Lisp integer specified by *arg*. The C data type of the return value, `intmax_t`, is the widest integer data type supported by the C compiler, typically `long long`. If the value of *arg* doesn't fit into an `intmax_t`, the function signals an error using the error symbol `overflow-error`.

`bool extract_big_integer (emacs_env *env, emacs_value arg, int *sign, ptrdiff_t *count, emacs_limb_t *magnitude)` [Function]
: This function, which is available since Emacs 27, extracts the integer value of *arg*. The value of *arg* must be an integer (fixnum or bignum). If *sign* is not `NULL`, it stores the sign of *arg* (-1, 0, or +1) into `*sign`. The magnitude is stored into *magnitude* as follows. If *count* and *magnitude* are both non-`NULL`, then *magnitude* must point to an array of at least `*count` `unsigned long` elements. If *magnitude* is large enough to hold the magnitude of *arg*, then this function writes the magnitude into the *magnitude* array in little-endian form, stores the number of array elements written into `*count`, and returns `true`. If *magnitude* is not large enough, it stores the required array size into `*count`, signals an error, and returns `false`. If *count* is not `NULL` and *magnitude* is `NULL`, then the function stores the required array size into `*count` and returns `true`.

  Emacs guarantees that the maximum required value of `*count` never exceeds `min (PTRDIFF_MAX, SIZE_MAX) / sizeof (emacs_limb_t)`, so you can use `malloc (*count * sizeof *magnitude)` to allocate the `magnitude` array without worrying about integer overflow in the size calculation.

`emacs_limb_t` [Type alias]
: This is an unsigned integer type, used as the element type for the magnitude arrays for the big integer conversion functions. The type is guaranteed to have unique object representations, i.e., no padding bits.

`EMACS_LIMB_MAX` [Macro]
: This macro expands to a constant expression specifying the maximum possible value for an `emacs_limb_t` object. The expression is suitable for use in `#if`.

`double extract_float (emacs_env *env, emacs_value arg)` [Function]

This function returns the value of a Lisp float specified by *arg*, as a C `double` value.

`struct timespec extract_time (emacs_env *env, emacs_value arg)` [Function]

This function, which is available since Emacs 27, interprets *arg* as an Emacs Lisp time value and returns the corresponding `struct timespec`. See Section 42.5 [Time of Day], page 1275. `struct timespec` represents a timestamp with nanosecond precision. It has the following members:

`time_t tv_sec`
: Whole number of seconds.

`long tv_nsec`
: Fractional seconds as a number of nanoseconds. For timestamps returned by `extract_time`, this is always nonnegative and less than one billion. (Although POSIX requires the type of `tv_nsec` to be `long`, the type is `long long` on some nonstandard platforms.)

See Section "Elapsed Time" in `libc`.

If *time* has higher precision than nanoseconds, then this function truncates it to nanosecond precision towards negative infinity. This function signals an error if *time* (truncated to nanoseconds) cannot be represented by `struct timespec`. For example, if `time_t` is a 32-bit integer type, then a *time* value of ten billion seconds would signal an error, but a *time* value of 600 picoseconds would get truncated to zero.

If you need to deal with time values that are not representable by `struct timespec`, or if you want higher precision, call the Lisp function `encode-time` and work with its return value. See Section 42.7 [Time Conversion], page 1278.

`bool copy_string_contents (emacs_env *env, emacs_value arg, char *buf, ptrdiff_t *len)` [Function]

This function stores the UTF-8 encoded text of a Lisp string specified by *arg* in the array of `char` pointed by *buf*, which should have enough space to hold at least `*len` bytes, including the terminating null byte. The argument *len* must not be a `NULL` pointer, and, when the function is called, it should point to a value that specifies the size of *buf* in bytes.

If the buffer size specified by `*len` is large enough to hold the string's text, the function stores in `*len` the actual number of bytes copied to *buf*, including the terminating null byte, and returns `true`. If the buffer is too small, the function raises the `args-out-of-range` error condition, stores the required number of bytes in `*len`, and returns `false`. See Section E.8.5 [Module Nonlocal], page 1380, for how to handle pending error conditions.

The argument *buf* can be a `NULL` pointer, in which case the function stores in `*len` the number of bytes required for storing the contents of *arg*, and returns `true`. This is how you can determine the size of *buf* needed to store a particular string: first call `copy_string_contents` with `NULL` as *buf*, then allocate enough memory to hold the number of bytes stored by the function in `*len`, and call the function again with non-`NULL` *buf* to actually perform the text copying.

`emacs_value vec_get (emacs_env *env, emacs_value vector, ptrdiff_t index)` [Function]

This function returns the element of *vector* at *index*. The *index* of the first vector element is zero. The function raises the `args-out-of-range` error condition if the value of *index* is invalid. To extract C data from the value the function returns, use the other extraction functions described here, as appropriate for the Lisp data type stored in that element of the vector.

`ptrdiff_t vec_size (emacs_env *env, emacs_value vector)` [Function]

This function returns the number of elements in *vector*.

`void vec_set (emacs_env *env, emacs_value vector, ptrdiff_t index, emacs_value value)` [Function]

This function stores *value* in the element of *vector* whose index is *index*. It raises the `args-out-of-range` error condition if the value of *index* is invalid.

The following API functions create `emacs_value` objects from basic C data types. They all return the created `emacs_value` object.

`emacs_value make_integer (emacs_env *env, intmax_t n)` [Function]

This function takes an integer argument *n* and returns the corresponding `emacs_value` object. It returns either a fixnum or a bignum depending on whether the value of *n* is inside the limits set by `most-negative-fixnum` and `most-positive-fixnum` (see Section 3.1 [Integer Basics], page 38).

`emacs_value make_big_integer (emacs_env *env, int sign, ptrdiff_t count, const emacs_limb_t *magnitude)` [Function]

This function, which is available since Emacs 27, takes an arbitrary-sized integer argument and returns a corresponding `emacs_value` object. The *sign* argument gives the sign of the return value. If *sign* is nonzero, then *magnitude* must point to an array of at least *count* elements specifying the little-endian magnitude of the return value.

The following example uses the GNU Multiprecision Library (GMP) to calculate the next probable prime after a given integer. See `gmp`, for a general overview of GMP, and see Section "Integer Import and Export" in `gmp` for how to convert the `magnitude` array to and from GMP `mpz_t` values.

```
#include <emacs-module.h>
int plugin_is_GPL_compatible;

#include <assert.h>
#include <limits.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>

#include <gmp.h>

static void
```

```
memory_full (emacs_env *env)
{
  static const char message[] = "Memory exhausted";
  emacs_value data = env->make_string (env, message,
                                       strlen (message));
  env->non_local_exit_signal
    (env, env->intern (env, "error"),
     env->funcall (env, env->intern (env, "list"), 1, &data));
}

enum
{
  order = -1, endian = 0, nails = 0,
  limb_size = sizeof (emacs_limb_t),
  max_nlimbs = ((SIZE_MAX < PTRDIFF_MAX ? SIZE_MAX : PTRDIFF_MAX)
                / limb_size)
};

static bool
extract_big_integer (emacs_env *env, emacs_value arg, mpz_t result)
{
  ptrdiff_t nlimbs;
  bool ok = env->extract_big_integer (env, arg, NULL, &nlimbs, NULL);
  if (!ok)
    return false;
  assert (0 < nlimbs && nlimbs <= max_nlimbs);
  emacs_limb_t *magnitude = malloc (nlimbs * limb_size);
  if (magnitude == NULL)
    {
      memory_full (env);
      return false;
    }
  int sign;
  ok = env->extract_big_integer (env, arg, &sign, &nlimbs, magnitude);
  assert (ok);
  mpz_import (result, nlimbs, order, limb_size, endian, nails, magnitude);
  free (magnitude);
  if (sign < 0)
    mpz_neg (result, result);
  return true;
}

static emacs_value
make_big_integer (emacs_env *env, const mpz_t value)
{
  size_t nbits = mpz_sizeinbase (value, 2);
  int bitsperlimb = CHAR_BIT * limb_size - nails;
```

```
  size_t nlimbs = nbits / bitsperlimb + (nbits % bitsperlimb != 0);
  emacs_limb_t *magnitude
    = nlimbs <= max_nlimbs ? malloc (nlimbs * limb_size) : NULL;
  if (magnitude == NULL)
    {
      memory_full (env);
      return NULL;
    }
  size_t written;
  mpz_export (magnitude, &written, order, limb_size, endian, nails, value);
  assert (written == nlimbs);
  assert (nlimbs <= PTRDIFF_MAX);
  emacs_value result = env->make_big_integer (env, mpz_sgn (value),
                                              nlimbs, magnitude);
  free (magnitude);
  return result;
}

static emacs_value
next_prime (emacs_env *env, ptrdiff_t nargs, emacs_value *args,
            void *data)
{
  assert (nargs == 1);
  mpz_t p;
  mpz_init (p);
  extract_big_integer (env, args[0], p);
  mpz_nextprime (p, p);
  emacs_value result = make_big_integer (env, p);
  mpz_clear (p);
  return result;
}

int
emacs_module_init (struct emacs_runtime *runtime)
{
  emacs_env *env = runtime->get_environment (runtime);
  emacs_value symbol = env->intern (env, "next-prime");
  emacs_value func
    = env->make_function (env, 1, 1, next_prime, NULL, NULL);
  emacs_value args[] = {symbol, func};
  env->funcall (env, env->intern (env, "defalias"), 2, args);
  return 0;
}
```

`emacs_value make_float (emacs_env *env, double d)` [Function]

This function takes a `double` argument d and returns the corresponding Emacs floating-point value.

`emacs_value make_time (emacs_env *env, struct timespec time)` [Function]

This function, which is available since Emacs 27, takes a `struct timespec` argument *time* and returns the corresponding Emacs timestamp as a pair `(ticks . hz)`. See Section 42.5 [Time of Day], page 1275. The return value represents exactly the same timestamp as *time*: all input values are representable, and there is never a loss of precision. `time.tv_sec` and `time.tv_nsec` can be arbitrary values. In particular, there's no requirement that *time* be normalized. This means that `time.tv_nsec` can be negative or larger than 999,999,999.

`emacs_value make_string (emacs_env *env, const char *str, ptrdiff_t len)` [Function]

This function creates an Emacs string from C text string pointed by *str* whose length in bytes, not including the terminating null byte, is *len*. The original string in *str* can be either an ASCII string or a UTF-8 encoded non-ASCII string; it can include embedded null bytes, and doesn't have to end in a terminating null byte at `str[len]`. The function raises the `overflow-error` error condition if *len* is negative or exceeds the maximum length of an Emacs string. If *len* is zero, then *str* can be `NULL`, otherwise it must point to valid memory. For nonzero *len*, `make_string` returns unique mutable string objects.

`emacs_value make_unibyte_string (emacs_env *env, const char *str, ptrdiff_t len)` [Function]

This function is like `make_string`, but has no restrictions on the values of the bytes in the C string, and can be used to pass binary data to Emacs in the form of a unibyte string.

The API does not provide functions to manipulate Lisp data structures, for example, create lists with `cons` and `list` (see Section 5.4 [Building Lists], page 82), extract list members with `car` and `cdr` (see Section 5.3 [List Elements], page 79), create vectors with `vector` (see Section 6.5 [Vector Functions], page 118), etc. For these, use `intern` and `funcall`, described in the next subsection, to call the corresponding Lisp functions.

Normally, `emacs_value` objects have a rather short lifetime: it ends when the `emacs_env` pointer used for their creation goes out of scope. Occasionally, you may need to create *global references*: `emacs_value` objects that live as long as you wish. Use the following two functions to manage such objects.

`emacs_value make_global_ref (emacs_env *env, emacs_value value)` [Function]

This function returns a global reference for *value*.

`void free_global_ref (emacs_env *env, emacs_value global_value)` [Function]

This function frees the *global_value* previously created by `make_global_ref`. The *global_value* is no longer valid after the call. Your module code should pair each call to `make_global_ref` with the corresponding `free_global_ref`.

An alternative to keeping around C data structures that need to be passed to module functions later is to create *user pointer* objects. A user pointer, or `user-ptr`, object is a Lisp

object that encapsulates a C pointer and can have an associated finalizer function, which is called when the object is garbage-collected (see Section E.3 [Garbage Collection], page 1355). The module API provides functions to create and access `user-ptr` objects. These functions raise the `wrong-type-argument` error condition if they are called on `emacs_value` that doesn't represent a `user-ptr` object.

`emacs_value make_user_ptr (emacs_env *env, emacs_finalizer fin, void *ptr)` [Function]

This function creates and returns a `user-ptr` object which wraps the C pointer *ptr*. The finalizer function *fin* can be a `NULL` pointer (meaning no finalizer), or it can be a function of the following signature:

```
typedef void (*emacs_finalizer) (void *ptr);
```

If *fin* is not a `NULL` pointer, it will be called with the *ptr* as the argument when the `user-ptr` object is garbage-collected. Don't run any expensive code in a finalizer, because GC must finish quickly to keep Emacs responsive.

`void *get_user_ptr (emacs_env *env, emacs_value arg)` [Function]

This function extracts the C pointer from the Lisp object represented by *arg*.

`void set_user_ptr (emacs_env *env, emacs_value arg, void *ptr)` [Function]

This function sets the C pointer embedded in the `user-ptr` object represented by *arg* to *ptr*.

`emacs_finalizer get_user_finalizer (emacs_env *env, emacs_value arg)` [Function]

This function returns the finalizer of the `user-ptr` object represented by *arg*, or `NULL` if it doesn't have a finalizer.

`void set_user_finalizer (emacs_env *env, emacs_value arg, emacs_finalizer fin)` [Function]

This function changes the finalizer of the `user-ptr` object represented by *arg* to be *fin*. If *fin* is a `NULL` pointer, the `user-ptr` object will have no finalizer.

Note that the `emacs_finalizer` type works for both user pointer an module function finalizers. See [Module Function Finalizers], page 1371.

E.8.4 Miscellaneous Convenience Functions for Modules

This subsection describes a few convenience functions provided by the module API. Like the functions described in previous subsections, all of them are actually function pointers, and need to be called via the `emacs_env` pointer. Description of functions that were introduced after Emacs 25 calls out the first version where they became available.

`bool eq (emacs_env *env, emacs_value a, emacs_value b)` [Function]

This function returns `true` if the Lisp objects represented by *a* and *b* are identical, `false` otherwise. This is the same as the Lisp function `eq` (see Section 2.8 [Equality Predicates], page 34), but avoids the need to intern the objects represented by the arguments.

There are no API functions for other equality predicates, so you will need to use `intern` and `funcall`, described below, to perform more complex equality tests.

`bool is_not_nil (emacs_env *env, emacs_value arg)` [Function]
This function tests whether the Lisp object represented by *arg* is non-`nil`; it returns `true` or `false` accordingly.

Note that you could implement an equivalent test by using `intern` to get an `emacs_value` representing `nil`, then use `eq`, described above, to test for equality. But using this function is more convenient.

`emacs_value type_of (emacs_env *env, emacs_value arg)` [Function]
This function returns the type of *arg* as a value that represents a symbol: `string` for a string, `integer` for an integer, `process` for a process, etc. See Section 2.7 [Type Predicates], page 31. You can use `intern` and `eq` to compare against known type symbols, if your code needs to depend on the object type.

`emacs_value intern (emacs_env *env, const char *name)` [Function]
This function returns an interned Emacs symbol whose name is *name*, which should be an ASCII null-terminated string. It creates a new symbol if one does not already exist.

Together with `funcall`, described below, this function provides a means for invoking any Lisp-callable Emacs function, provided that its name is a pure ASCII string. For example, here's how to intern a symbol whose name `name_str` is non-ASCII, by calling the more powerful Emacs `intern` function (see Section 9.3 [Creating Symbols], page 135):

```
emacs_value fintern = env->intern (env, "intern");
emacs_value sym_name =
  env->make_string (env, name_str, strlen (name_str));
emacs_value symbol = env->funcall (env, fintern, 1, &sym_name);
```

`emacs_value funcall (emacs_env *env, emacs_value func,` [Function]
`ptrdiff_t nargs, emacs_value *args)`
This function calls the specified *func* passing it *nargs* arguments from the array pointed to by *args*. The argument *func* can be a function symbol (e.g., returned by `intern` described above), a module function returned by `make_function` (see Section E.8.2 [Module Functions], page 1368), a subroutine written in C, etc. If *nargs* is zero, *args* can be a `NULL` pointer.

The function returns the value that *func* returned.

If your module includes potentially long-running code, it is a good idea to check from time to time in that code whether the user wants to quit, e.g., by typing `C-g` (see Section 22.11 [Quitting], page 472). The following function, which is available since Emacs 26.1, is provided for that purpose.

`bool should_quit (emacs_env *env)` [Function]
This function returns `true` if the user wants to quit. In that case, we recommend that your module function aborts any on-going processing and returns as soon as possible. In most cases, use `process_input` instead.

To process input events in addition to checking whether the user wants to quit, use the following function, which is available since Emacs 27.1.

`enum emacs_process_input_result process_input (emacs_env *env)` [Function]

This function processes pending input events. It returns `emacs_process_input_quit` if the user wants to quit or an error occurred while processing signals. In that case, we recommend that your module function aborts any on-going processing and returns as soon as possible. If the module code may continue running, `process_input` returns `emacs_process_input_continue`. The return value is `emacs_process_input_continue` if and only if there is no pending nonlocal exit in `env`. If the module continues after calling `process_input`, global state such as variable values and buffer content may have been modified in arbitrary ways.

`int open_channel (emacs_env *env, emacs_value pipe_process)` [Function]

This function, which is available since Emacs 28, opens a channel to an existing pipe process. *pipe_process* must refer to an existing pipe process created by `make-pipe-process`. [Pipe Processes], page 1092. If successful, the return value will be a new file descriptor that you can use to write to the pipe. Unlike all other module functions, you can use the returned file descriptor from arbitrary threads, even if no module environment is active. You can use the `write` function to write to the file descriptor. Once done, close the file descriptor using `close`. Section "Low-Level I/O" in `libc`.

E.8.5 Nonlocal Exits in Modules

Emacs Lisp supports nonlocal exits, whereby program control is transferred from one point in a program to another remote point. See Section 11.7 [Nonlocal Exits], page 177. Thus, Lisp functions called by your module might exit nonlocally by calling `signal` or `throw`, and your module functions must handle such nonlocal exits properly. Such handling is needed because C programs will not automatically release resources and perform other cleanups in these cases; your module code must itself do it. The module API provides facilities for that, described in this subsection. They are generally available since Emacs 25; those of them that became available in later releases explicitly call out the first Emacs version where they became part of the API.

When some Lisp code called by a module function signals an error or throws, the nonlocal exit is trapped, and the pending exit and its associated data are stored in the environment. Whenever a nonlocal exit is pending in the environment, any module API function called with a pointer to that environment will return immediately without any processing (the functions `non_local_exit_check`, `non_local_exit_get`, and `non_local_exit_clear` are exceptions from this rule). If your module function then does nothing and returns to Emacs, a pending nonlocal exit will cause Emacs to act on it: signal an error or throw to the corresponding `catch`.

So the simplest "handling" of nonlocal exits in module functions is to do nothing special and let the rest of your code to run as if nothing happened. However, this can cause two classes of problems:

- Your module function might use uninitialized or undefined values, since API functions return immediately without producing the expected results.
- Your module might leak resources, because it might not have the opportunity to release them.

Therefore, we recommend that your module functions check for nonlocal exit conditions and recover from them, using the functions described below.

`enum emacs_funcall_exit non_local_exit_check (emacs_env *env)` [Function]

This function returns the kind of nonlocal exit condition stored in *env*. The possible values are:

`emacs_funcall_exit_return`
: The last API function exited normally.

`emacs_funcall_exit_signal`
: The last API function signaled an error.

`emacs_funcall_exit_throw`
: The last API function exited via `throw`.

`enum emacs_funcall_exit non_local_exit_get (emacs_env *env, emacs_value *symbol, emacs_value *data)` [Function]

This function returns the kind of nonlocal exit condition stored in *env*, like `non_local_exit_check` does, but it also returns the full information about the nonlocal exit, if any. If the return value is `emacs_funcall_exit_signal`, the function stores the error symbol in `*symbol` and the error data in `*data` (see Section 11.7.3.1 [Signaling Errors], page 180). If the return value is `emacs_funcall_exit_throw`, the function stores the `catch` tag symbol in `*symbol` and the `throw` value in `*data`. The function doesn't store anything in memory pointed by these arguments when the return value is `emacs_funcall_exit_return`.

You should check nonlocal exit conditions where it matters: before you allocated some resource or after you allocated a resource that might need freeing, or where a failure means further processing is impossible or infeasible.

Once your module function detected that a nonlocal exit is pending, it can either return to Emacs (after performing the necessary local cleanup), or it can attempt to recover from the nonlocal exit. The following API functions will help with these tasks.

`void non_local_exit_clear (emacs_env *env)` [Function]

This function clears the pending nonlocal exit conditions and data from *env*. After calling it, the module API functions will work normally. Use this function if your module function can recover from nonlocal exits of the Lisp functions it calls and continue, and also before calling any of the following two functions (or any other API functions, if you want them to perform their intended processing when a nonlocal exit is pending).

`void non_local_exit_throw (emacs_env *env, emacs_value tag, emacs_value value)` [Function]

This function throws to the Lisp `catch` symbol represented by *tag*, passing it *value* as the value to return. Your module function should in general return soon after calling this function. One use of this function is when you want to re-throw a non-local exit from one of the called API or Lisp functions.

`void non_local_exit_signal (emacs_env *env, emacs_value symbol, emacs_value data)` [Function]

This function signals the error represented by the error symbol *symbol* with the specified error data *data*. The module function should return soon after calling this function. This function could be useful, e.g., for signaling errors from module functions to Emacs.

E.9 Object Internals

Emacs Lisp provides a rich set of the data types. Some of them, like cons cells, integers and strings, are common to nearly all Lisp dialects. Some others, like markers and buffers, are quite special and needed to provide the basic support to write editor commands in Lisp. To implement such a variety of object types and provide an efficient way to pass objects between the subsystems of an interpreter, there is a set of C data structures and a special type to represent the pointers to all of them, which is known as *tagged pointer*.

In C, the tagged pointer is an object of type `Lisp_Object`. Any initialized variable of such a type always holds the value of one of the following basic data types: integer, symbol, string, cons cell, float, or vectorlike object. Each of these data types has the corresponding tag value. All tags are enumerated by `enum Lisp_Type` and placed into a 3-bit bitfield of the `Lisp_Object`. The rest of the bits is the value itself. Integers are immediate, i.e., directly represented by those *value bits*, and all other objects are represented by the C pointers to a corresponding object allocated from the heap. Width of the `Lisp_Object` is platform- and configuration-dependent: usually it's equal to the width of an underlying platform pointer (i.e., 32-bit on a 32-bit machine and 64-bit on a 64-bit one), but also there is a special configuration where `Lisp_Object` is 64-bit but all pointers are 32-bit. The latter trick was designed to overcome the limited range of values for Lisp integers on a 32-bit system by using 64-bit `long long` type for `Lisp_Object`.

The following C data structures are defined in `lisp.h` to represent the basic data types beyond integers:

`struct Lisp_Cons`
: Cons cell, an object used to construct lists.

`struct Lisp_String`
: String, the basic object to represent a sequence of characters.

`struct Lisp_Vector`
: Array, a fixed-size set of Lisp objects which may be accessed by an index.

`struct Lisp_Symbol`
: Symbol, the unique-named entity commonly used as an identifier.

`struct Lisp_Float`
: Floating-point value.

These types are the first-class citizens of an internal type system. Since the tag space is limited, all other types are the subtypes of `Lisp_Vectorlike`. Vector subtypes are enumerated by `enum pvec_type`, and nearly all complex objects like windows, buffers, frames, and processes fall into this category.

Below there is a description of a few subtypes of `Lisp_Vectorlike`. Buffer object represents the text to display and edit. Window is the part of display structure which shows the buffer or is used as a container to recursively place other windows on the same frame. (Do not confuse Emacs Lisp window object with the window as an entity managed by the user interface system like X; in Emacs terminology, the latter is called frame.) Finally, process object is used to manage the subprocesses.

E.9.1 Buffer Internals

Two structures (see `buffer.h`) are used to represent buffers in C. The `buffer_text` structure contains fields describing the text of a buffer; the `buffer` structure holds other fields. In the case of indirect buffers, two or more `buffer` structures reference the same `buffer_text` structure.

Here are some of the fields in `struct buffer_text`:

`beg` The address of the buffer contents. The buffer contents is a linear C array of `char`, with the gap somewhere in its midst.

`gpt`
`gpt_byte` The character and byte positions of the buffer gap. See Section 28.13 [Buffer Gap], page 691.

`z`
`z_byte` The character and byte positions of the end of the buffer text.

`gap_size` The size of buffer's gap. See Section 28.13 [Buffer Gap], page 691.

`modiff`
`save_modiff`
`chars_modiff`
`overlay_modiff`
These fields count the number of buffer-modification events performed in this buffer. `modiff` is incremented after each buffer-modification event, and is never otherwise changed; `save_modiff` contains the value of `modiff` the last time the buffer was visited or saved; `chars_modiff` counts only modifications to the characters in the buffer, ignoring all other kinds of changes (such as text properties); and `overlay_modiff` counts only modifications to the buffer's overlays.

`beg_unchanged`
`end_unchanged`
The number of characters at the start and end of the text that are known to be unchanged since the last complete redisplay.

`unchanged_modified`
`overlay_unchanged_modified`
The values of `modiff` and `overlay_modiff`, respectively, after the last complete redisplay. If their current values match `modiff` or `overlay_modiff`, that means `beg_unchanged` and `end_unchanged` contain no useful information.

`markers` The markers that refer to this buffer. This is actually a single marker, and successive elements in its marker *chain* (a linked list) are the other markers referring to this buffer text.

`intervals`
: The interval tree which records the text properties of this buffer.

Some of the fields of `struct buffer` are:

`header` A header of type `union vectorlike_header` is common to all vectorlike objects.

`own_text` A `struct buffer_text` structure that ordinarily holds the buffer contents. In indirect buffers, this field is not used.

`text` A pointer to the `buffer_text` structure for this buffer. In an ordinary buffer, this is the `own_text` field above. In an indirect buffer, this is the `own_text` field of the base buffer.

`next` A pointer to the next buffer, in the chain of all buffers, including killed buffers. This chain is used only for allocation and garbage collection, in order to collect killed buffers properly.

`pt`
`pt_byte` The character and byte positions of point in a buffer.

`begv`
`begv_byte`
: The character and byte positions of the beginning of the accessible range of text in the buffer.

`zv`
`zv_byte` The character and byte positions of the end of the accessible range of text in the buffer.

`base_buffer`
: In an indirect buffer, this points to the base buffer. In an ordinary buffer, it is null.

`local_flags`
: This field contains flags indicating that certain variables are local in this buffer. Such variables are declared in the C code using `DEFVAR_PER_BUFFER`, and their buffer-local bindings are stored in fields in the buffer structure itself. (Some of these fields are described in this table.)

`modtime` The modification time of the visited file. It is set when the file is written or read. Before writing the buffer into a file, this field is compared to the modification time of the file to see if the file has changed on disk. See Section 28.5 [Buffer Modification], page 679.

`auto_save_modified`
: The time when the buffer was last auto-saved.

`last_window_start`
: The `window-start` position in the buffer as of the last time the buffer was displayed in a window.

`clip_changed`
: This flag indicates that narrowing has changed in the buffer. See Section 31.4 [Narrowing], page 867.

`prevent_redisplay_optimizations_p`
: This flag indicates that redisplay optimizations should not be used to display this buffer.

`inhibit_buffer_hooks`
: This flag indicates that the buffer should not run the hooks `kill-buffer-hook`, `kill-buffer-query-functions` (see Section 28.10 [Killing Buffers], page 687), and `buffer-list-update-hook` (see Section 28.8 [Buffer List], page 683). It is set at buffer creation (see Section 28.9 [Creating Buffers], page 686), and avoids slowing down internal or temporary buffers, such as those created by `with-temp-buffer` (see [Current Buffer], page 675).

`name`
: A Lisp string that names the buffer. It is guaranteed to be unique. See Section 28.3 [Buffer Names], page 676. This and the following fields have their names in the C struct definition end in a `_` to indicate that they should not be accessed directly, but via the `BVAR` macro, like this:

```
Lisp_Object buf_name = BVAR (buffer, name);
```

`save_length`
: The length of the file this buffer is visiting, when last read or saved. It can have 2 special values: -1 means auto-saving was turned off in this buffer, and -2 means don't turn off auto-saving if buffer text shrinks a lot. This and other fields concerned with saving are not kept in the `buffer_text` structure because indirect buffers are never saved.

`directory`
: The directory for expanding relative file names. This is the value of the buffer-local variable `default-directory` (see Section 26.9.4 [File Name Expansion], page 641).

`filename`
: The name of the file visited in this buffer, or `nil`. This is the value of the buffer-local variable `buffer-file-name` (see Section 28.4 [Buffer File Name], page 677).

`undo_list`
`backed_up`
`auto_save_file_name`
`auto_save_file_format`
`read_only`
`file_format`
`file_truename`
`invisibility_spec`
`display_count`
`display_time`
: These fields store the values of Lisp variables that are automatically buffer-local (see Section 12.11 [Buffer-Local Variables], page 207), whose corresponding variable names have the additional prefix `buffer-` and have underscores replaced with dashes. For instance, `undo_list` stores the value of `buffer-undo-list`.

`mark`
: The mark for the buffer. The mark is a marker, hence it is also included on the list `markers`. See Section 32.7 [The Mark], page 875.

`local_var_alist`
: The association list describing the buffer-local variable bindings of this buffer, not including the built-in buffer-local bindings that have special slots in the buffer object. (Those slots are omitted from this table.) See Section 12.11 [Buffer-Local Variables], page 207.

`major_mode`
: Symbol naming the major mode of this buffer, e.g., `lisp-mode`.

`mode_name`
: Pretty name of the major mode, e.g., `"Lisp"`.

`keymap`
`abbrev_table`
`syntax_table`
`category_table`
`display_table`
: These fields store the buffer's local keymap (see Chapter 23 [Keymaps], page 481), abbrev table (see Section 38.1 [Abbrev Tables], page 1069), syntax table (see Chapter 36 [Syntax Tables], page 1025), category table (see Section 36.8 [Categories], page 1039), and display table (see Section 41.23.2 [Display Tables], page 1248).

`downcase_table`
`upcase_table`
`case_canon_table`
: These fields store the conversion tables for converting text to lower case, upper case, and for canonicalizing text for case-fold search. See Section 4.10 [Case Tables], page 74.

`minor_modes`
: An alist of the minor modes of this buffer.

`pt_marker`
`begv_marker`
`zv_marker`
: These fields are only used in an indirect buffer, or in a buffer that is the base of an indirect buffer. Each holds a marker that records `pt`, `begv`, and `zv` respectively, for this buffer when the buffer is not current.

`mode_line_format`
`header_line_format`
`case_fold_search`
`tab_width`
`fill_column`
`left_margin`
`auto_fill_function`
`truncate_lines`
`word_wrap`
`ctl_arrow`
`bidi_display_reordering`
`bidi_paragraph_direction`
`selective_display`
`selective_display_ellipses`
`overwrite_mode`
`abbrev_mode`
`mark_active`
`enable_multibyte_characters`
`buffer_file_coding_system`
`cache_long_line_scans`
`point_before_scroll`
`left_fringe_width`
`right_fringe_width`
`fringes_outside_margins`
`scroll_bar_width`
`indicate_empty_lines`
`indicate_buffer_boundaries`
`fringe_indicator_alist`
`fringe_cursor_alist`
`scroll_up_aggressively`
`scroll_down_aggressively`
`cursor_type`
`cursor_in_non_selected_windows`
: These fields store the values of Lisp variables that are automatically buffer-local (see Section 12.11 [Buffer-Local Variables], page 207), whose corresponding variable names have underscores replaced with dashes. For instance, `mode_line_format` stores the value of `mode-line-format`.

`overlays` The interval tree containing this buffer's overlays.

`last_selected_window`
: This is the last window that was selected with this buffer in it, or `nil` if that window no longer displays this buffer.

E.9.2 Window Internals

The fields of a window (for a complete list, see the definition of `struct window` in `window.h`) include:

`frame` The frame that this window is on, as a Lisp object.

`mini` Non-zero if this window is a minibuffer window, a window showing the minibuffer or the echo area.

`pseudo_window_p`
Non-zero if this window is a *pseudo window*. A pseudo window is either a window used to display the menu bar or the tool bar (when Emacs uses toolkits that don't display their own menu bar and tool bar) or the tab bar or a window showing a tooltip on a tooltip frame. Pseudo windows are in general not accessible from Lisp code.

`parent` Internally, Emacs arranges windows in a tree; each group of siblings has a parent window whose area includes all the siblings. This field points to the window's parent in that tree, as a Lisp object. For the root window of the tree and a minibuffer window this is always `nil`.

Parent windows do not display buffers, and play little role in display except to shape their child windows. Emacs Lisp programs cannot directly manipulate parent windows; they operate on the windows at the leaves of the tree, which actually display buffers.

`contents` For a leaf window and windows showing a tooltip, this is the buffer, as a Lisp object, that the window is displaying. For an internal ("parent") window, this is its first child window. For a pseudo window showing a menu or tool bar this is `nil`. It is also `nil` for a window that has been deleted.

`next`
`prev` The next and previous sibling of this window as Lisp objects. `next` is `nil` if the window is the right-most or bottom-most in its group; `prev` is `nil` if it is the left-most or top-most in its group. Whether the sibling is left/right or up/down is determined by the `horizontal` field of the sibling's parent: if it's non-zero, the siblings are arranged horizontally.

As a special case, `next` of a frame's root window points to the frame's minibuffer window, provided this is not a minibuffer-only or minibuffer-less frame. On such frames `prev` of the minibuffer window points to that frame's root window. In any other case, the root window's `next` and the minibuffer window's (if present) `prev` fields are `nil`.

`left_col` The left-hand edge of the window, measured in columns, relative to the leftmost column (column 0) of the window's native frame.

`top_line` The top edge of the window, measured in lines, relative to the topmost line (line 0) of the window's native frame.

`pixel_left`
`pixel_top`
The left-hand and top edges of this window, measured in pixels, relative to the top-left corner (0, 0) of the window's native frame.

`total_cols`
`total_lines`
The total width and height of the window, measured in columns and lines respectively. The values include scroll bars and fringes, dividers and/or the separator line on the right of the window (if any).

`pixel_width;`
`pixel_height;`
: The total width and height of the window measured in pixels.

`start`
: A marker pointing to the position in the buffer that is the first character (in the logical order, see Section 41.27 [Bidirectional Display], page 1255) displayed in the window.

`pointm`
: This is the value of point in the current buffer when this window is selected; when it is not selected, it retains its previous value.

`old_pointm`
: The value of `pointm` at the last redisplay time.

`force_start`
: If this flag is non-`nil`, it says that the window has been scrolled explicitly by the Lisp program, and the value of the window's `start` was set for redisplay to honor. This affects what the next redisplay does if point is off the screen: instead of scrolling the window to show the text around point, it moves point to a location that is on the screen.

`optional_new_start`
: This is similar to `force_start`, but the next redisplay will only obey it if point stays visible.

`start_at_line_beg`
: Non-`nil` means current value of `start` was the beginning of a line when it was chosen.

`use_time`
: This is the last time that the window was selected. The function `get-lru-window` uses this field.

`sequence_number`
: A unique number assigned to this window when it was created.

`last_modified`
: The `modiff` field of the window's buffer, as of the last time a redisplay completed in this window.

`last_overlay_modified`
: The `overlay_modiff` field of the window's buffer, as of the last time a redisplay completed in this window.

`last_point`
: The buffer's value of point, as of the last time a redisplay completed in this window.

`last_had_star`
: A non-zero value means the window's buffer was modified when the window was last updated.

`vertical_scroll_bar_type`
`horizontal_scroll_bar_type`
: The types of this window's vertical and horizontal scroll bars.

`scroll_bar_width`
`scroll_bar_height`
: The width of this window's vertical scroll bar and the height of this window's horizontal scroll bar, in pixels.

`left_margin_cols`
`right_margin_cols`
: The widths of the left and right margins in this window. A value of zero means no margin.

`left_fringe_width`
`right_fringe_width`
: The pixel widths of the left and right fringes in this window. A value of -1 means use the values of the frame.

`fringes_outside_margins`
: A non-zero value means the fringes outside the display margins; othersize they are between the margin and the text.

`window_end_pos`
: This is computed as `z` minus the buffer position of the last glyph in the current matrix of the window. The value is only valid if `window_end_valid` is non-zero.

`window_end_bytepos`
: The byte position corresponding to `window_end_pos`.

`window_end_vpos`
: The window-relative vertical position of the line containing `window_end_pos`.

`window_end_valid`
: This field is set to a non-zero value if `window_end_pos` and `window_end_vpos` are truly valid. This is zero if nontrivial redisplay is pre-empted, since in that case the display that `window_end_pos` was computed for did not get onto the screen.

`cursor`
: A structure describing where the cursor is in this window.

`last_cursor_vpos`
: The window-relative vertical position of the line showing the cursor as of the last redisplay that finished.

`phys_cursor`
: A structure describing where the cursor of this window physically is.

`phys_cursor_type`
`phys_cursor_height`
`phys_cursor_width`
: The type, height, and width of the cursor that was last displayed on this window.

`phys_cursor_on_p`
: This field is non-zero if the cursor is physically on.

`cursor_off_p`
: Non-zero means the cursor in this window is logically off. This is used for blinking the cursor.

`last_cursor_off_p`
: This field contains the value of `cursor_off_p` as of the time of the last redisplay.

`must_be_updated_p`
: This is set to 1 during redisplay when this window must be updated.

`hscroll`
: This is the number of columns that the display in the window is scrolled horizontally to the left. Normally, this is 0. When only the current line is hscrolled, this describes how much the current line is scrolled.

`min_hscroll`
: Minimum value of `hscroll`, set by the user via `set-window-hscroll` (see Section 29.23 [Horizontal Scrolling], page 770). When only the current line is hscrolled, this describes the horizontal scrolling of lines other than the current one.

`vscroll`
: Vertical scroll amount, in pixels. Normally, this is 0.

`dedicated`
: Non-`nil` if this window is dedicated to its buffer.

`combination_limit`
: This window's combination limit, meaningful only for a parent window. If this is `t`, then it is not allowed to delete this window and recombine its child windows with other siblings of this window.

`window_parameters`
: The alist of this window's parameters.

`display_table`
: The window's display table, or `nil` if none is specified for it.

`update_mode_line`
: Non-zero means this window's mode line needs to be updated.

`mode_line_height`
`header_line_height`
: The height in pixels of the mode line and the header line, or −1 if not known.

`base_line_number`
: The line number of a certain position in the buffer, or zero. This is used for displaying the line number of point in the mode line.

`base_line_pos`
: The position in the buffer for which the line number is known, or zero meaning none is known. If it is −1, don't display the line number as long as the window shows that buffer.

`column_number_displayed`
: The column number currently displayed in this window's mode line, or −1 if column numbers are not being displayed.

`current_matrix`
`desired_matrix`
: Glyph matrices describing the current and desired display of this window.

E.9.3 Process Internals

The fields of a process (for a complete list, see the definition of `struct Lisp_Process` in `process.h`) include:

`name` A Lisp string, the name of the process.

`command` A list containing the command arguments that were used to start this process. For a network or serial process, it is `nil` if the process is running or `t` if the process is stopped.

`filter` A Lisp function used to accept output from the process.

`sentinel` A Lisp function called whenever the state of the process changes.

`buffer` The associated buffer of the process.

`pid` An integer, the operating system's process ID. Pseudo-processes such as network or serial connections use a value of 0.

`childp` A flag, `t` if this is really a child process. For a network or serial connection, it is a plist based on the arguments to `make-network-process` or `make-serial-process`.

`mark` A marker indicating the position of the end of the last output from this process inserted into the buffer. This is often but not always the end of the buffer.

`kill_without_query`
: If this is non-zero, killing Emacs while this process is still running does not ask for confirmation about killing the process.

`raw_status`
: The raw process status, as returned by the `wait` system call.

`status` The process status, as `process-status` should return it. This is a Lisp symbol, a cons cell, or a list.

`tick`
`update_tick`
: If these two fields are not equal, a change in the status of the process needs to be reported, either by running the sentinel or by inserting a message in the process buffer.

`pty_flag` Non-zero if communication with the subprocess uses a pty; zero if it uses a pipe.

`infd` The file descriptor for input from the process.

`outfd` The file descriptor for output to the process.

`tty_name` The name of the terminal that the subprocess is using, or `nil` if it is using pipes.

`decode_coding_system`
: Coding-system for decoding the input from this process.

`decoding_buf`
: A working buffer for decoding.

`decoding_carryover`
: Size of carryover in decoding.

`encode_coding_system`
: Coding-system for encoding the output to this process.

`encoding_buf`
: A working buffer for encoding.

`inherit_coding_system_flag`
: Flag to set `coding-system` of the process buffer from the coding system used to decode process output.

`type`
: Symbol indicating the type of process: `real`, `network`, `serial`.

E.10 C Integer Types

Here are some guidelines for use of integer types in the Emacs C source code. These guidelines sometimes give competing advice; common sense is advised.

- Avoid arbitrary limits. For example, avoid `int len = strlen (s);` unless the length of `s` is required for other reasons to fit in `int` range.
- Do not assume that signed integer arithmetic wraps around on overflow. This is no longer true of Emacs porting targets: signed integer overflow has undefined behavior in practice, and can dump core or even cause earlier or later code to behave illogically. Unsigned overflow does wrap around reliably, modulo a power of two.
- Prefer signed types to unsigned, as code gets confusing when signed and unsigned types are combined. Many other guidelines assume that types are signed; in the rarer cases where unsigned types are needed, similar advice may apply to the unsigned counterparts (e.g., `size_t` instead of `ptrdiff_t`, or `uintptr_t` instead of `intptr_t`).
- Prefer `int` for Emacs character codes, in the range 0 .. 0x3FFFFF. More generally, prefer `int` for integers known to be in `int` range, e.g., screen column counts.
- Prefer `ptrdiff_t` for sizes, i.e., for integers bounded by the maximum size of any individual C object or by the maximum number of elements in any C array. This is part of Emacs's general preference for signed types. Using `ptrdiff_t` limits objects to `PTRDIFF_MAX` bytes, but larger objects would cause trouble anyway since they would break pointer subtraction, so this does not impose an arbitrary limit.
- Avoid `ssize_t` except when communicating to low-level APIs that have `ssize_t`-related limitations. Although it's equivalent to `ptrdiff_t` on typical platforms, `ssize_t` is occasionally narrower, so using it for size-related calculations could overflow. Also, `ptrdiff_t` is more ubiquitous and better-standardized, has standard `printf` formats, and is the basis for Emacs's internal size-overflow checking. When using `ssize_t`, please note that POSIX requires support only for values in the range -1 .. `SSIZE_MAX`.
- Normally, prefer `intptr_t` for internal representations of pointers, or for integers bounded only by the number of objects that can exist at any given time or by the total number of bytes that can be allocated. However, prefer `uintptr_t` to represent pointer arithmetic that could cross page boundaries. For example, on a machine with a 32-bit address space an array could cross the 0x7fffffff/0x80000000 boundary, which would cause an integer overflow when adding 1 to `(intptr_t) 0x7fffffff`.

- Prefer the Emacs-defined type `EMACS_INT` for representing values converted to or from Emacs Lisp fixnums, as fixnum arithmetic is based on `EMACS_INT`.
- When representing a system value (such as a file size or a count of seconds since the Epoch), prefer the corresponding system type (e.g., `off_t`, `time_t`). Do not assume that a system type is signed, unless this assumption is known to be safe. For example, although `off_t` is always signed, `time_t` need not be.
- Prefer `intmax_t` for representing values that might be any signed integer value. A `printf`-family function can print such a value via a format like `"%"PRIdMAX`.
- Prefer `bool`, `false` and `true` for booleans. Using `bool` can make programs easier to read and a bit faster than using `int`. Although it is also OK to use `int`, `0` and `1`, this older style is gradually being phased out. When using `bool`, respect the limitations of the replacement implementation of `bool`. In particular, boolean bitfields should be of type `bool_bf`, not `bool`, so that they work correctly even when compiling Objective C with standard GCC.
- In bitfields, prefer `unsigned int` or `signed int` to `int`, as `int` is less portable: it might be signed, and might not be. Single-bit bit fields should be `unsigned int` or `bool_bf` so that their values are 0 or 1.

Appendix F Standard Errors

Here is a list of the more important error symbols in standard Emacs, grouped by concept. The list includes each symbol's message and a cross reference to a description of how the error can occur.

Each error symbol has a set of parent error conditions that is a list of symbols. Normally this list includes the error symbol itself and the symbol `error`. Occasionally it includes additional symbols, which are intermediate classifications, narrower than `error` but broader than a single error symbol. For example, all the errors in accessing files have the condition `file-error`. If we do not say here that a certain error symbol has additional error conditions, that means it has none.

As a special exception, the error symbols `quit` and `minibuffer-quit` don't have the condition `error`, because quitting is not considered an error.

Most of these error symbols are defined in C (mainly `data.c`), but some are defined in Lisp. For example, the file `userlock.el` defines the `file-locked` and `file-supersession` errors. Several of the specialized Lisp libraries distributed with Emacs define their own error symbols. We do not attempt to list of all those here.

See Section 11.7.3 [Errors], page 179, for an explanation of how errors are generated and handled.

`error` The message is '`error`'. See Section 11.7.3 [Errors], page 179.

`quit` The message is '`Quit`'. See Section 22.11 [Quitting], page 472.

`minibuffer-quit`
: The message is '`Quit`'. This is a subcategory of `quit`. See Section 22.11 [Quitting], page 472.

`args-out-of-range`
: The message is '`Args out of range`'. This happens when trying to access an element beyond the range of a sequence, buffer, or other container-like object. See Chapter 6 [Sequences Arrays Vectors], page 102, and see Chapter 33 [Text], page 880.

`arith-error`
: The message is '`Arithmetic error`'. This occurs when trying to perform integer division by zero. See Section 3.5 [Numeric Conversions], page 43, and see Section 3.6 [Arithmetic Operations], page 45.

`beginning-of-buffer`
: The message is '`Beginning of buffer`'. See Section 31.2.1 [Character Motion], page 856.

`buffer-read-only`
: The message is '`Buffer is read-only`'. See Section 28.7 [Read Only Buffers], page 682.

`circular-list`
: The message is '`List contains a loop`'. This happens when a circular structure is encountered. See Section 2.6 [Circular Objects], page 30.

`cl-assertion-failed`
: The message is '`Assertion failed`'. This happens when the `cl-assert` macro fails a test. See Section "Assertions" in *Common Lisp Extensions*.

`coding-system-error`
: The message is '`Invalid coding system`'. See Section 34.10.3 [Lisp and Coding Systems], page 982.

`cyclic-function-indirection`
: The message is '`Symbol's chain of function indirections contains a loop`'. See Section 10.1.4 [Function Indirection], page 147.

`cyclic-variable-indirection`
: The message is '`Symbol's chain of variable indirections contains a loop`'. See Section 12.15 [Variable Aliases], page 222.

`dbus-error`
: The message is '`D-Bus error`'. See Section "Errors and Events" in *D-Bus integration in Emacs*.

`end-of-buffer`
: The message is '`End of buffer`'. See Section 31.2.1 [Character Motion], page 856.

`end-of-file`
: The message is '`End of file during parsing`'. Note that this is not a subcategory of `file-error`, because it pertains to the Lisp reader, not to file I/O. See Section 20.3 [Input Functions], page 375.

`file-already-exists`
: This is a subcategory of `file-error`. See Section 26.4 [Writing to Files], page 618.

`permission-denied`
: This is a subcategory of `file-error`, which occurs when the OS doesn't allow Emacs to access a file or a directory for some reason.

`file-date-error`
: This is a subcategory of `file-error`. It occurs when `copy-file` tries and fails to set the last-modification time of the output file. See Section 26.7 [Changing Files], page 631.

`file-error`
: We do not list the error-strings of this error and its subcategories, because the error message is normally constructed from the data items alone when the error condition `file-error` is present. Thus, the error-strings are not very relevant. However, these error symbols do have `error-message` properties, and if no data is provided, the `error-message` property *is* used. See Chapter 26 [Files], page 610.

`file-missing`
: This is a subcategory of `file-error`. It occurs when an operation attempts to act on a file that is missing. See Section 26.7 [Changing Files], page 631.

`compression-error`
: This is a subcategory of `file-error`, which results from problems handling a compressed file. See Section 16.1 [How Programs Do Loading], page 299.

`file-locked`
: This is a subcategory of `file-error`. See Section 26.5 [File Locks], page 619.

`file-supersession`
: This is a subcategory of `file-error`. See Section 28.6 [Modification Time], page 680.

`file-notify-error`
: This is a subcategory of `file-error`. It happens, when a file could not be watched for changes. See Section 42.20 [File Notifications], page 1301.

`remote-file-error`
: This is a subcategory of `file-error`, which results from problems in accessing a remote file. See Section "Remote Files" in *The GNU Emacs Manual*. Often, this error appears when timers, process filters, process sentinels or special events in general try to access a remote file, and collide with another remote file operation. In general it is a good idea to write a bug report. See Section "Bugs" in *The GNU Emacs Manual*.

`ftp-error`
: This is a subcategory of `remote-file-error`, which results from problems in accessing a remote file using ftp. See Section "Remote Files" in *The GNU Emacs Manual*.

`invalid-function`
: The message is '`Invalid function`'. See Section 10.1.4 [Function Indirection], page 147.

`invalid-read-syntax`
: The message is usually '`Invalid read syntax`'. See Section 2.1 [Printed Representation], page 8. This error can also be raised by commands like `eval-expression` when there's text following an expression. In that case, the message is '`Trailing garbage following expression`'.

`invalid-regexp`
: The message is '`Invalid regexp`'. See Section 35.3 [Regular Expressions], page 999.

`mark-inactive`
: The message is '`The mark is not active now`'. See Section 32.7 [The Mark], page 875.

`no-catch`
: The message is '`No catch for tag`'. See Section 11.7.1 [Catch and Throw], page 177.

`range-error`
: The message is `Arithmetic range error`.

`overflow-error`
: The message is '`Arithmetic overflow error`'. This is a subcategory of `range-error`. This can happen with integers exceeding the `integer-width` limit. See Section 3.1 [Integer Basics], page 38.

`scan-error`
: The message is '`Scan error`'. This happens when certain syntax-parsing functions find invalid syntax or mismatched parentheses. Conventionally raised with three argument: a human-readable error message, the start of the obstacle that cannot be moved over, and the end of the obstacle. See Section 31.2.6 [List Motion], page 863, and see Section 36.6 [Parsing Expressions], page 1033.

`search-failed`
: The message is '`Search failed`'. See Chapter 35 [Searching and Matching], page 996.

`setting-constant`
: The message is '`Attempt to set a constant symbol`'. This happens when attempting to assign values to `nil`, `t`, `most-positive-fixnum`, `most-negative-fixnum`, and keyword symbols. It also happens when attempting to assign values to `enable-multibyte-characters` and some other symbols whose direct assignment is not allowed for some reason. See Section 12.2 [Constant Variables], page 189.

`text-read-only`
: The message is '`Text is read-only`'. This is a subcategory of `buffer-read-only`. See Section 33.19.4 [Special Properties], page 927.

`undefined-color`
: The message is '`Undefined color`'. See Section 30.23 [Color Names], page 848.

`user-error`
: The message is the empty string. See Section 11.7.3.1 [Signaling Errors], page 180.

`user-search-failed`
: This is like '`search-failed`', but doesn't trigger the debugger, like '`user-error`'. See Section 11.7.3.1 [Signaling Errors], page 180, and see Chapter 35 [Searching and Matching], page 996. This is used for searching in Info files, see Section "Search Text" in *Info*.

`void-function`
: The message is '`Symbol's function definition is void`'. See Section 13.9 [Function Cells], page 248.

`void-variable`
: The message is '`Symbol's value as variable is void`'. See Section 12.7 [Accessing Variables], page 197.

`wrong-number-of-arguments`
: The message is '`Wrong number of arguments`'. See Section 13.2.3 [Argument List], page 234.

`wrong-type-argument`
: The message is '`Wrong type argument`'. See Section 2.7 [Type Predicates], page 31.

`unknown-image-type`
: The message is '`Cannot determine image type`'. See Section 41.17 [Images], page 1211.

`inhibited-interaction`
: The message is '`User interaction while inhibited`'. This error is signaled when `inhibit-interaction` is non-`nil` and a user interaction function (like `read-from-minibuffer`) is called.

Appendix G Standard Keymaps

In this section we list some of the more general keymaps. Many of these exist when Emacs is first started, but some are loaded only when the respective feature is accessed.

There are many other, more specialized, maps than these; in particular those associated with major and minor modes. The minibuffer uses several keymaps (see Section 21.6.3 [Completion Commands], page 401). For more details on keymaps, see Chapter 23 [Keymaps], page 481.

`2C-mode-map`
: A sparse keymap for subcommands of the prefix `C-x 6`.
See Section "Two-Column Editing" in *The GNU Emacs Manual*.

`abbrev-map`
: A sparse keymap for subcommands of the prefix `C-x a`.
See Section "Defining Abbrevs" in *The GNU Emacs Manual*.

`button-buffer-map`
: A sparse keymap useful for buffers containing buffers.
You may want to use this as a parent keymap. See Section 41.20 [Buttons], page 1236.

`button-map`
: A sparse keymap used by buttons.

`ctl-x-4-map`
: A sparse keymap for subcommands of the prefix `C-x 4`.

`ctl-x-5-map`
: A sparse keymap for subcommands of the prefix `C-x 5`.

`ctl-x-map`
: A full keymap for `C-x` commands.

`ctl-x-r-map`
: A sparse keymap for subcommands of the prefix `C-x r`.
See Section "Registers" in *The GNU Emacs Manual*.

`esc-map` A full keymap for `ESC` (or `Meta`) commands.

`function-key-map`
: The parent keymap of all `local-function-key-map` (q.v.) instances.

`global-map`
: The full keymap containing default global key bindings.
Modes should not modify the Global map.

`goto-map` A sparse keymap used for the `M-g` prefix key.

`help-map` A sparse keymap for the keys following the help character `C-h`.
See Section 25.6 [Help Functions], page 604.

`Helper-help-map`
: A full keymap used by the help utility package.
It has the same keymap in its value cell and in its function cell.

`input-decode-map`
: The keymap for translating keypad and function keys.
If there are none, then it contains an empty sparse keymap. See Section 23.15 [Translation Keymaps], page 505.

`key-translation-map`
: A keymap for translating keys. This one overrides ordinary key bindings, unlike `local-function-key-map`. See Section 23.15 [Translation Keymaps], page 505.

`kmacro-keymap`
: A sparse keymap for keys that follows the *C-x C-k* prefix search.
See Section "Keyboard Macros" in *The GNU Emacs Manual*.

`local-function-key-map`
: The keymap for translating key sequences to preferred alternatives.
If there are none, then it contains an empty sparse keymap. See Section 23.15 [Translation Keymaps], page 505.

`menu-bar-file-menu`
`menu-bar-edit-menu`
`menu-bar-options-menu`
`global-buffers-menu-map`
`menu-bar-tools-menu`
`menu-bar-help-menu`
: These keymaps display the main, top-level menus in the menu bar.
Some of them contain sub-menus. For example, the Edit menu contains `menu-bar-search-menu`, etc. See Section 23.18.5 [Menu Bar], page 517.

`minibuffer-inactive-mode-map`
: A full keymap used in the minibuffer when it is not active.
See Section "Editing in the Minibuffer" in *The GNU Emacs Manual*.

`mode-line-coding-system-map`
`mode-line-input-method-map`
`mode-line-column-line-number-mode-map`
: These keymaps control various areas of the mode line.
See Section 24.4 [Mode Line Format], page 551.

`mode-specific-map`
: The keymap for characters following *C-c*. Note, this is in the global map. This map is not actually mode-specific: its name was chosen to be informative in *C-h b* (`display-bindings`), where it describes the main use of the *C-c* prefix key.

`mouse-appearance-menu-map`
: A sparse keymap used for the *S-mouse-1* key.

`mule-keymap`
: The global keymap used for the *C-x* `RET` prefix key.

`narrow-map`
: A sparse keymap for subcommands of the prefix *C-x n*.

`prog-mode-map`
: The keymap used by Prog mode.
See Section 24.2.5 [Basic Major Modes], page 537.

`query-replace-map`
`multi-query-replace-map`
: A sparse keymap used for responses in `query-replace` and related commands; also for `y-or-n-p` and `map-y-or-n-p`. The functions that use this map do not support prefix keys; they look up one event at a time. `multi-query-replace-map` extends `query-replace-map` for multi-buffer replacements. See Section 35.7 [Search and Replace], page 1019.

`search-map`
: A sparse keymap that provides global bindings for search-related commands.

`special-mode-map`
: The keymap used by Special mode.
See Section 24.2.5 [Basic Major Modes], page 537.

`tab-prefix-map`
: The global keymap used for the `C-x t` prefix key for tab-bar related commands.
See Section "Tab Bars" in *The GNU Emacs Manual*.

`tab-bar-map`
: The keymap defining the contents of the tab bar.
See Section "Tab Bars" in *The GNU Emacs Manual*.

`tool-bar-map`
: The keymap defining the contents of the tool bar.
See Section 23.18.6 [Tool Bar], page 518.

`universal-argument-map`
: A sparse keymap used while processing `C-u`.
See Section 22.12 [Prefix Command Arguments], page 474.

`vc-prefix-map`
: The global keymap used for the `C-x v` prefix key.

`x-alternatives-map`
: A sparse keymap used to map certain keys under graphical frames.
The function `x-setup-function-keys` uses this.

Appendix H Standard Hooks

The following is a list of some hook variables that let you provide functions to be called from within Emacs on suitable occasions.

Most of these variables have names ending with '`-hook`'. They are *normal hooks*, run by means of `run-hooks`. The value of such a hook is a list of functions; the functions are called with no arguments and their values are completely ignored. The recommended way to put a new function on such a hook is to call `add-hook`. See Section 24.1 [Hooks], page 525, for more information about using hooks.

The variables whose names end in '`-functions`' are usually *abnormal hooks* (some old code may also use the deprecated '`-hooks`' suffix). Their values are lists of functions, but these functions are called in a special way: they are either passed arguments, or their return values are used in some way. The variables whose names end in '`-function`' have single functions as their values.

This is not an exhaustive list, it only covers the more general hooks. For example, every major mode defines a hook named '*modename*`-mode-hook`'. The major mode command runs this normal hook with `run-mode-hooks` as the very last thing it does. See Section 24.2.6 [Mode Hooks], page 538. Most minor modes have mode hooks too.

A special feature allows you to specify expressions to evaluate if and when a file is loaded (see Section 16.10 [Hooks for Loading], page 314). That feature is not exactly a hook, but does a similar job.

`activate-mark-hook`
`deactivate-mark-hook`
: See Section 32.7 [The Mark], page 875.

`after-change-functions`
`before-change-functions`
`first-change-hook`
: See Section 33.34 [Change Hooks], page 964.

`after-change-major-mode-hook`
`change-major-mode-after-body-hook`
: See Section 24.2.6 [Mode Hooks], page 538.

`after-init-hook`
`before-init-hook`
`emacs-startup-hook`
`window-setup-hook`
: See Section 42.1.2 [Init File], page 1263.

`after-insert-file-functions`
`write-region-annotate-functions`
`write-region-post-annotation-function`
: See Section 26.13 [Format Conversion], page 656.

`after-make-frame-functions`
`before-make-frame-hook`
`server-after-make-frame-hook`
: See Section 30.1 [Creating Frames], page 788.

`after-save-hook`
`before-save-hook`
`write-contents-functions`
`write-file-functions`
: See Section 26.2 [Saving Buffers], page 614.

`after-setting-font-hook`
: Hook run after a frame's font changes.

`auto-save-hook`
: See Section 27.2 [Auto-Saving], page 666.

`before-hack-local-variables-hook`
`hack-local-variables-hook`
: See Section 12.12 [File Local Variables], page 213.

`buffer-access-fontify-functions`
: See Section 33.19.7 [Lazy Properties], page 936.

`buffer-list-update-hook`
: Hook run when the buffer list changes (see Section 28.8 [Buffer List], page 683).

`buffer-quit-function`
: Function to call to quit the current buffer.

`change-major-mode-hook`
: See Section 12.11.2 [Creating Buffer-Local], page 208.

`comint-password-function`
: This abnormal hook permits a derived mode to supply a password for the underlying command interpreter without prompting the user.

`command-line-functions`
: See Section 42.1.4 [Command-Line Arguments], page 1266.

`delayed-warnings-hook`
: The command loop runs this soon after `post-command-hook` (q.v.).

`focus-in-hook`
`focus-out-hook`
: See Section 30.10 [Input Focus], page 827.

`delete-frame-functions`
`after-delete-frame-functions`
: See Section 30.7 [Deleting Frames], page 824.

`delete-terminal-functions`
: See Section 30.2 [Multiple Terminals], page 789.

`pop-up-frame-function`
`split-window-preferred-function`
: See Section 29.13.4 [Choosing Window Options], page 738.

`echo-area-clear-hook`
: See Section 41.4.4 [Echo Area Customization], page 1142.

`find-file-hook`
`find-file-not-found-functions`
: See Section 26.1.1 [Visiting Functions], page 610.

`font-lock-extend-after-change-region-function`
: See Section 24.6.9.2 [Region to Refontify], page 579.

`font-lock-extend-region-functions`
: See Section 24.6.9 [Multiline Font Lock], page 577.

`font-lock-fontify-buffer-function`
`font-lock-fontify-region-function`
`font-lock-mark-block-function`
`font-lock-unfontify-buffer-function`
`font-lock-unfontify-region-function`
: See Section 24.6.4 [Other Font Lock Variables], page 572.

`fontification-functions`
: See Section 41.12.7 [Automatic Face Assignment], page 1182.

`frame-auto-hide-function`
: See Section 29.16 [Quitting Windows], page 752.

`quit-window-hook`
: See Section 29.16 [Quitting Windows], page 752.

`kill-buffer-hook`
`kill-buffer-query-functions`
: See Section 28.10 [Killing Buffers], page 687.

`kill-emacs-hook`
`kill-emacs-query-functions`
: See Section 42.2.1 [Killing Emacs], page 1267.

`menu-bar-update-hook`
: See Section 23.18.5 [Menu Bar], page 517.

`minibuffer-setup-hook`
`minibuffer-exit-hook`
: See Section 21.15 [Minibuffer Misc], page 422.

`mouse-leave-buffer-hook`
: Hook run when the user mouse-clicks in a window.

`mouse-position-function`
: See Section 30.16 [Mouse Position], page 838.

`prefix-command-echo-keystrokes-functions`
: An abnormal hook run by prefix commands (such as `C-u`) which should return a string describing the current prefix state. For example, `C-u` produces '`C-u-`' and '`C-u 1 2 3-`'. Each hook function is called with no arguments and should return a string describing the current prefix state, or `nil` if there's no prefix state. See Section 22.12 [Prefix Command Arguments], page 474.

`prefix-command-preserve-state-hook`
: Hook run when a prefix command needs to preserve the prefix by passing the current prefix command state to the next command. For example, `C-u` needs to pass the state to the next command when the user types `C-u -` or follows `C-u` with a digit.

`pre-redisplay-functions`
: Hook run in each window just before redisplaying it. See Section 41.2 [Forcing Redisplay], page 1133.

`post-command-hook`
`pre-command-hook`
: See Section 22.1 [Command Overview], page 424.

`post-gc-hook`
: See Section E.3 [Garbage Collection], page 1355.

`post-self-insert-hook`
: See Section 24.3.2 [Keymaps and Minor Modes], page 547.

`suspend-hook`
`suspend-resume-hook`
`suspend-tty-functions`
`resume-tty-functions`
: See Section 42.2.2 [Suspending Emacs], page 1268.

`syntax-begin-function`
`syntax-propertize-extend-region-functions`
`syntax-propertize-function`
`font-lock-syntactic-face-function`
: See Section 24.6.8 [Syntactic Font Lock], page 577. See Section 36.4 [Syntax Properties], page 1032.

`temp-buffer-setup-hook`
`temp-buffer-show-function`
`temp-buffer-show-hook`
: See Section 41.8 [Temporary Displays], page 1150.

`tty-setup-hook`
: See Section 42.1.3 [Terminal-Specific], page 1265.

`window-configuration-change-hook`
`window-scroll-functions`
`window-size-change-functions`
: See Section 29.28 [Window Hooks], page 782.

Index

"

‘"’ in printing 378
‘"’ in strings 20

#

‘##’ read syntax 15
‘#$’ 320
‘#'’ syntax 244
‘#(’ read syntax 21
‘#:’ read syntax 15
‘#^’ read syntax 22
‘#@*count*’ 320
‘#*n*#’ read syntax 30
‘#*n*=’ read syntax 30

$

‘$’ in display 1134
‘$’ in regexp 1002

%

% 47
‘%’ in format 67

&

‘&’ in replacement 1015
&optional 234
&rest 234

'

‘'’ for quoting 151

(

‘(’ in regexp 1005
‘(...)’ in lists 16
‘(?:’ in regexp 1005
(emacs_env 1378

)

‘)’ in regexp 1005

*

* 46
‘*’ in interactive 426
‘*’ in regexp 999
scratch 534

+

+ 45
‘+’ in regexp 1000

,

, (with backquote) 152
,@ (with backquote) 152

—

- 46
--disable-build-details
  option to configure 1352
--enable-locallisppath option
  to configure 1352
--temacs option, and dumping method 1351

.

‘.’ in lists 18
‘.’ in regexp 999
.#, lock file names 619
.elpaignore file 1311
.emacs 1263

/

/ 46
/= 43
/dev/tty 1125

:

:deferred, JSONRPC keyword 962
:documentation 236
:equal 1059
:match 1059
:notification-dispatcher 960
:pred 1059
:repeat 486
:request-dispatcher 960

;

‘;’ for commenting 10

<

< 43
<= 43

=

= 42

>

> 43
>= 43

?

‘?’ in character constant 11
? in minibuffer 391
‘?’ in regexp 1000

[

‘[’ in regexp 1000
[...] (Edebug) 363

]

‘]’ in regexp 1000

^

‘^’ in interactive 426
‘^’ in regexp 1001

`

` 151
‘ (list substitution) 151

@

‘@’ in interactive 426

\

‘\’ in character constant 12
‘\’ in display 1134
‘\’ in printing 378
‘\’ in regexp 1002
‘\’ in replacement 1015
‘\’ in strings 20
‘\’ in symbols 14
‘\'’ in regexp 1007
\(in strings 864
‘\<’ in regexp 1007
‘\=’ in regexp 1007
‘\>’ in regexp 1007
‘_<’ in regexp 1007
‘_>’ in regexp 1007
‘\`’ in regexp 1007
‘\a’ 12
‘\b’ 12
‘\b’ in regexp 1007
‘\B’ in regexp 1007
‘\e’ 12
‘\f’ 12
‘\n’ 12
‘\n’ in print 381
‘\n’ in replacement 1015
‘\r’ 12
‘\s’ 12
‘\s’ in regexp 1006
‘\S’ in regexp 1007
‘\t’ 12
‘\v’ 12
‘\w’ in regexp 1006
‘\W’ in regexp 1006

|

‘|’ in regexp 1005

1

1+ 45
1- 45
1value 370

2

2C-mode-map 489
2D box 1170

3

3D box 1170

A

abbrev ... 1069
abbrev properties ... 1074
abbrev table properties ... 1075
abbrev tables ... 1069
abbrev tables in modes ... 531
`abbrev-all-caps` ... 1073
`abbrev-expand-function` ... 1073
`abbrev-expansion` ... 1072
`abbrev-file-name` ... 1071
`abbrev-get` ... 1074
`abbrev-insert` ... 1072
`abbrev-map` ... 1400
`abbrev-minor-mode-table-alist` ... 1074
`abbrev-prefix-mark` ... 1072
`abbrev-put` ... 1074
`abbrev-start-location` ... 1073
`abbrev-start-location-buffer` ... 1073
`abbrev-symbol` ... 1072
`abbrev-table-get` ... 1075
`abbrev-table-name-list` ... 1070
`abbrev-table-p` ... 1069
`abbrev-table-put` ... 1075
`abbreviate-file-name` ... 641
abbreviated file names ... 641
abbrevs, looking up and expanding ... 1072
`abbrevs-changed` ... 1071
abnormal hook ... 525
`abort-recursive-edit` ... 477
aborting ... 476
`abs` ... 43
absolute edges ... 797
absolute file name ... 639
absolute frame edges ... 797
absolute frame position ... 797
absolute position ... 797
accept input from processes ... 1107
`accept-change-group` ... 963
`accept-process-output` ... 1107
access control list ... 629
access minibuffer contents ... 421
`access-file` ... 622
accessibility of a file ... 621
accessible portion (of a buffer) ... 867
`accessible-keymaps` ... 508
accessing documentation strings ... 598
accessing hash tables ... 129
accessing plist properties ... 100
ACL entries ... 629
`acos` ... 51
`action` (button property) ... 1237
action alist for buffer display ... 733
action function, for buffer display ... 728
`action`, customization keyword ... 293
`activate-change-group` ... 963
`activate-mark-hook` ... 878
active display table ... 1249
active keymap ... 490
active keymap, controlling ... 492
active minibuffer ... 387
`active-minibuffer-window` ... 420
`ad-activate` ... 258
`adaptive-fill-first-line-regexp` ... 907
`adaptive-fill-function` ... 907
`adaptive-fill-mode` ... 906
`adaptive-fill-regexp` ... 907
`add-display-text-property` ... 1203
`add-face-text-property` ... 923
`add-function` ... 253
`add-hook` ... 527
`add-name-to-file` ... 631
`add-text-properties` ... 922
`add-to-history` ... 394
`add-to-invisibility-spec` ... 1147
`add-to-list` ... 86
`add-to-ordered-list` ... 87
`add-variable-watcher` ... 200
address field of register ... 16
`adjust-window-trailing-edge` ... 706
adjusting point ... 440
advertised binding ... 601
`advertised-calling-convention` (`declare` spec) ... 263
advice, add and remove ... 253
`advice-add` ... 256
`advice-eval-interactive-spec` ... 255
`advice-function-mapc` ... 255
`advice-function-member-p` ... 255
`advice-mapc` ... 257
`advice-member-p` ... 257
`advice-remove` ... 257
advices, porting from `defadvice` ... 258
advising functions ... 252
advising named functions ... 255
AEAD cipher ... 949
`affixation-function`, in completion ... 411
after-change notifier, for tree-sitter parse-tree ... 1048
`after-change-functions` ... 964
`after-change-major-mode-hook` ... 539
`after-delete-frame-functions` ... 824
`after-find-file` ... 613
`after-focus-change-function` ... 829
`after-init-hook` ... 1264
`after-init-time` ... 1261
`after-insert-file-functions` ... 660
`after-load-functions` ... 315
`after-make-frame-functions` ... 788
`after-pdump-load-hook` ... 1353
`after-revert-hook` ... 671
`after-save-hook` ... 616
`after-setting-font-hook` ... 1404
`after-string` (overlay property) ... 1160
alias, for coding systems ... 981
alias, for faces ... 1182
alias, for functions ... 238

alias, for variables ... 222
aligning header line, when line numbers are displayed ... 1207
alist ... 95
alist vs. plist ... 100
`alist-get` ... 96
`all-completions` ... 398
`all-threads` ... 1077
`allow-no-window`, a buffer display action alist entry ... 737
alnum character class, regexp ... 1003
alpha character class, regexp ... 1003
`alpha`, a frame parameter ... 821
`alpha-background`, a frame parameter ... 821
alt characters ... 14
alternative commands, defining ... 432
`always` ... 241
amalgamating commands, and undo ... 899
`amalgamating-undo-limit` ... 899
ancestor directory of file ... 649
`and` ... 162
animation ... 1229
`annotation-function`, in completion ... 411
anonymous face ... 1168
anonymous function ... 243
anonymous node, tree-sitter ... 1044
apostrophe for quoting ... 151
apostrophe, quoting in documentation strings ... 1344
`append` ... 83
`append-to-file` ... 618
`apply` ... 240
`apply`, and debugging ... 344
`apply-partially` ... 241
applying customizations ... 295
`apropos` ... 604
archive web server ... 1313
`aref` ... 116
`args`, customization keyword ... 291
argument ... 230
argument binding ... 234
argument lists, features ... 234
arguments for shell commands ... 1083
arguments, interactive entry ... 425
arguments, reading ... 386
`argv` ... 1267
`arith-error` example ... 184
`arith-error` in division ... 46
arithmetic operations ... 45
arithmetic shift ... 48
array ... 115
array elements ... 116
`arrayp` ... 116
ascii character class, regexp ... 1003
`ascii-case-table` ... 76
ASCII character codes ... 11
ASCII control characters ... 1247
`aset` ... 116
`ash` ... 48
`asin` ... 51
`ask-user-about-lock` ... 620
`ask-user-about-supersession-threat` ... 681
asking the user questions ... 414
`assoc` ... 95
`assoc-default` ... 97
`assoc-delete-all` ... 98
`assoc-string` ... 64
association list ... 95
`assq` ... 96
`assq-delete-all` ... 98
asynchronous native compilation, disable ... 332
asynchronous subprocess ... 1089
`atan` ... 51
`atom` ... 78
atomic changes ... 962
atomic windows ... 759
atoms ... 16
attributes of text ... 919
Auto Fill mode ... 907
`auto-coding-alist` ... 987
`auto-coding-functions` ... 988
`auto-coding-regexp-alist` ... 987
`auto-fill-chars` ... 908
`auto-fill-function` ... 908
`auto-hide-function`, a frame parameter ... 815
`auto-hscroll-mode` ... 771
`auto-lower`, a frame parameter ... 817
`auto-mode-alist` ... 534
`auto-raise`, a frame parameter ... 817
`auto-raise-tool-bar-buttons` ... 521
`auto-resize-tool-bars` ... 521
`auto-save-default` ... 668
`auto-save-file-name-p` ... 667
`auto-save-file-name-transforms` ... 666
`auto-save-hook` ... 668
`auto-save-interval` ... 668
`auto-save-list-file-name` ... 669
`auto-save-list-file-prefix` ... 669
`auto-save-mode` ... 666
`auto-save-timeout` ... 668
`auto-save-visited-file-name` ... 667
auto-selection of window ... 776
`auto-window-vscroll` ... 770
autoload ... 305
autoload by prefix ... 308
autoload cookie ... 306
autoload cookie, and safe values of variable ... 215
autoload errors ... 306
autoload object ... 231
autoload, when to use ... 308
`autoload-compute-prefixes` ... 308
`autoload-do-load` ... 308
`autoloadp` ... 306
automatic face assignment ... 1182
automatically buffer-local ... 207

B

`back-to-indentation` ... 918
`background-color`, a frame parameter ... 822
`background-mode`, a frame parameter ... 820
backing store ... 854
backquote (list substitution) ... 151
backquote-style patterns ... 171
backslash in character constants ... 12
backslash in regular expressions ... 1005
backslash in strings ... 20
backslash in symbols ... 14
backspace ... 12
`backtrace` ... 343
backtrace buffer ... 339
backtrace from emacsclient's `--eval` ... 335
backtrace of thread ... 1079
`backtrace-debug` ... 344
`backtrace-frame` ... 345
`backtrace-on-error-noninteractive` ... 343
`backtrace-on-redisplay-error` ... 337
backtracking ... 364
backtracking and POSIX regular expressions ... 1014
backtracking and regular expressions ... 999
backup file ... 661
backup files, rename or copy ... 663
`backup-buffer` ... 661
`backup-by-copying` ... 663
`backup-by-copying-when-linked` ... 663
`backup-by-copying-when-mismatch` ... 663
`backup-by-copying-when-privileged-mismatch` ... 663
`backup-directory-alist` ... 662
`backup-enable-predicate` ... 662
`backup-file-name-p` ... 665
`backup-inhibited` ... 662
backups and auto-saving ... 661
`backward-button` ... 1241
`backward-char` ... 857
`backward-delete-char-untabify` ... 889
`backward-delete-char-untabify-method` ... 889
`backward-list` ... 863
`backward-prefix-chars` ... 1033
`backward-sexp` ... 864
`backward-to-indentation` ... 918
`backward-up-list` ... 863
`backward-word` ... 857
`backward-word-strictly` ... 858
`balance-windows` ... 708
`balance-windows-area` ... 708
balanced parenthesis motion ... 863
balancing parentheses ... 1246
balancing window sizes ... 708
bare symbol ... 143
`bare-symbol` ... 144
`barf-if-buffer-read-only` ... 683
base 64 encoding ... 946
base buffer ... 689
base coding system ... 980
base direction of a paragraph ... 1256
base for reading an integer ... 38
base location, package archive ... 1312
base remapping, faces ... 1180
base type, in bindat specification ... 1128
`base64-decode-region` ... 946
`base64-decode-string` ... 947
`base64-encode-region` ... 946
`base64-encode-string` ... 946
`base64url-encode-region` ... 946
`base64url-encode-string` ... 946
basic code (of input character) ... 441
basic faces ... 1182
batch mode ... 1294
`batch-byte-compile` ... 319
`batch-native-compile` ... 329
baud, in serial connections ... 1127
`baud-rate` ... 1292
`beep` ... 1252
before point, insertion ... 884
`before-change-functions` ... 964
`before-hack-local-variables-hook` ... 215
`before-init-hook` ... 1264
`before-init-time` ... 1260
`before-make-frame-hook` ... 788
`before-revert-hook` ... 671
`before-save-hook` ... 616
`before-string` (overlay property) ... 1160
beginning of line ... 859
beginning of line in regexp ... 1002
`beginning-of-buffer` ... 858
`beginning-of-defun` ... 864
`beginning-of-defun-function` ... 864
`beginning-of-line` ... 859
bell ... 1252
bell character ... 12
`benchmark.el` ... 371
benchmarking ... 371
best practices ... 1336
`bidi-display-reordering` ... 1255
`bidi-find-overridden-directionality` ... 1258
`bidi-paragraph-direction` ... 1256
`bidi-paragraph-separate-re` ... 1256
`bidi-paragraph-start-re` ... 1256
`bidi-string-mark-left-to-right` ... 1258
bidirectional class of characters ... 972
bidirectional display ... 1255
bidirectional reordering ... 1255
big endian, in bindat specification ... 1128
bignum range ... 39
`bignump` ... 41
`binary` coding system ... 981
binary I/O in batch mode ... 376
bindat computed types ... 1131
bindat functions ... 1130
bindat packing and unpacking into arbitrary types ... 1131

bindat type expression ... 1128
bindat types ... 1128
bindat, define new type forms ... 1132
`bindat-defmacro` ... 1132
`bindat-get-field` ... 1130
`bindat-ip-to-string` ... 1131
`bindat-length` ... 1130
`bindat-pack` ... 1131
`bindat-type` ... 1128
`bindat-unpack` ... 1130
binding arguments ... 234
binding local variables ... 190
binding of a key ... 482
`bitmap-spec-p` ... 1172
bitmaps, fringe ... 1197
bitwise arithmetic ... 48
`blink-cursor-alist` ... 820
`blink-matching-delay` ... 1246
`blink-matching-open` ... 1246
`blink-matching-paren` ... 1246
`blink-matching-paren-distance` ... 1246
`blink-paren-function` ... 1246
blinking parentheses ... 1246
`bobp` ... 881
body height of a window ... 702
body of a window ... 693
body of function ... 233
body size of a window ... 703
body width of a window ... 703
`body-function`, a buffer display action alist entry ... 737
`bolp` ... 881
`bool-vector` ... 122
bool-vector length ... 102
`bool-vector-count-consecutive` ... 122
`bool-vector-count-population` ... 122
`bool-vector-exclusive-or` ... 122
`bool-vector-intersection` ... 122
`bool-vector-not` ... 122
`bool-vector-p` ... 122
`bool-vector-set-difference` ... 122
`bool-vector-subsetp` ... 122
`bool-vector-union` ... 122
Bool-vectors ... 121
boolean ... 2
`booleanp` ... 3
bootstrapping Emacs ... 1351
`border-color`, a frame parameter ... 822
`border-width`, a frame parameter ... 812
bottom dividers ... 1202
`bottom-divider`, prefix key ... 463
`bottom-divider-width`, a frame parameter ... 813
`bottom-visible`, a frame parameter ... 816
`boundp` ... 194
box diagrams, for lists ... 16
box face attribute, and `display` properties ... 1204
break ... 334
breakpoints (Edebug) ... 351
bucket (in obarray) ... 135
buffer ... 673
buffer boundaries, indicating ... 1195
buffer contents ... 880
buffer display ... 726
buffer display action alist ... 733
buffer display action function ... 728
buffer display action functions, precedence ... 741
buffer display conventions ... 745
buffer display display action ... 726
buffer file name ... 677
buffer gap ... 691
buffer input stream ... 372
buffer internals ... 1383
buffer list ... 683
buffer modification ... 679
buffer names ... 676
buffer output stream ... 376
buffer point changed by Edebug ... 358
buffer portion as string ... 881
buffer position ... 855
buffer text notation ... 4
buffer, read-only ... 682
`buffer-access-fontified-property` ... 936
`buffer-access-fontify-functions` ... 936
`buffer-auto-save-file-format` ... 659
`buffer-auto-save-file-name` ... 666
`buffer-backed-up` ... 661
`buffer-base-buffer` ... 690
buffer-button-map ... 1240
`buffer-chars-modified-tick` ... 680
`buffer-disable-undo` ... 901
`buffer-display-count` ... 722
`buffer-display-table` ... 1250
`buffer-display-time` ... 722
`buffer-enable-undo` ... 900
`buffer-end` ... 856
`buffer-file-coding-system` ... 981
`buffer-file-format` ... 658
`buffer-file-name` ... 677
`buffer-file-number` ... 678
`buffer-file-truename` ... 678
`buffer-hash` ... 948
`buffer-invisibility-spec` ... 1147
`buffer-list` ... 683
`buffer-list`, a frame parameter ... 814
`buffer-list-update-hook` ... 685, 1404
`buffer-list-update-hook` in temporary buffers ... 675
`buffer-live-p` ... 689
buffer-local variables ... 207
buffer-local variables in modes ... 531
`buffer-local-boundp` ... 210
`buffer-local-restore-state` ... 551
`buffer-local-set-state` ... 551
`buffer-local-value` ... 210
`buffer-local-variables` ... 210
`buffer-match-p` ... 685

`buffer-modified-p` ... 679
`buffer-modified-tick` ... 680
`buffer-name` ... 676
`buffer-name-history` ... 395
`buffer-narrowed-p` ... 868
`buffer-offer-save` ... 689
`buffer-predicate`, a frame parameter ... 814
`buffer-quit-function` ... 1404
`buffer-read-only` ... 682
`buffer-save-without-query` ... 689
`buffer-saved-size` ... 669
`buffer-size` ... 856
`buffer-stale-function` ... 671
`buffer-string` ... 882
`buffer-substring` ... 881
`buffer-substring-no-properties` ... 882
`buffer-substring-with-bidi-context` ... 1259
`buffer-swap-text` ... 690
`buffer-text-pixel-size` ... 1165
`buffer-undo-list` ... 898
`bufferp` ... 673
`bufferpos-to-filepos` ... 967
buffers to display on frame ... 814
buffers without undo information ... 676
buffers, controlled in windows ... 722
buffers, creating ... 686
buffers, killing ... 687
bugs ... 1
bugs in this manual ... 1
build details ... 1352
building Emacs ... 1351
building lists ... 82
built-in function ... 230
`bump-use-time`, a buffer display action alist entry ... 736
`bury-buffer` ... 684
`butlast` ... 81
`button` (button property) ... 1237
button buffer commands ... 1240
button properties ... 1237
button types ... 1238
`button-activate` ... 1239
`button-at` ... 1240
button-down event ... 447
`button-end` ... 1239
`button-face`, customization keyword ... 293
`button-get` ... 1239
`button-has-type-p` ... 1240
`button-label` ... 1239
`button-map` ... 1237
`button-prefix`, customization keyword ... 293
`button-put` ... 1239
`button-start` ... 1239
`button-suffix`, customization keyword ... 293
`button-type` ... 1239
`button-type-get` ... 1240
`button-type-put` ... 1240
`button-type-subtype-p` ... 1240
`buttonize` ... 1239
buttons in buffers ... 1236
byte compilation ... 317
byte compiler warnings, how to avoid ... 1342
byte packing and unpacking ... 1128
byte to string ... 969
`byte-boolean-vars` ... 224, 1364
byte-code ... 317
byte-code function ... 324
byte-code object ... 324
`byte-code-function-p` ... 232
`byte-compile` ... 318
`byte-compile` and `byte-optimize`, advising ... 260
`byte-compile-debug` ... 317
`byte-compile-dynamic` ... 321
`byte-compile-dynamic-docstrings` ... 320
`byte-compile-error-on-warn` ... 323
`byte-compile-file` ... 319
byte-compiler errors ... 322
byte-compiler warnings ... 322
byte-compiling macros ... 271
byte-compiling `require` ... 310
`byte-recompile-directory` ... 319
`byte-to-position` ... 967
`byte-to-string` ... 969
bytes ... 54
bytesize, in serial connections ... 1127

C

C programming language ... 1360
C-c ... 489
C-g ... 472
C-h ... 489
`C-M-x` ... 347
C-x ... 489
C-x 4 ... 489
C-x 5 ... 489
C-x 6 ... 489
`C-x C-a C-m` ... 348
C-x `RET` ... 489
C-x t ... 489
C-x v ... 489
`C-x X =` ... 357
`caaaar` ... 81
`caaadr` ... 81
`caaar` ... 81
`caadar` ... 81
`caaddr` ... 81
`caadr` ... 81
`caar` ... 81
`cadaar` ... 81
`cadadr` ... 81
`cadar` ... 81
`caddar` ... 81
`cadddr` ... 81
`caddr` ... 81
`cadr` ... 81

calendrical computations ... 1285
calendrical information ... 1278
call stack ... 343
`call-interactively` ... 434
`call-process` ... 1085
`call-process`, command-line arguments from minibuffer ... 1083
`call-process-region` ... 1087
`call-process-shell-command` ... 1088
`call-shell-region` ... 1088
`called-interactively-p` ... 436
calling a function ... 239
`cancel-change-group` ... 963
`cancel-debug-on-entry` ... 338
`cancel-debug-on-variable-change` ... 338
`cancel-timer` ... 1289
canonical character height ... 799
canonical character width ... 799
capitalization ... 73
`capitalize` ... 73
`capitalize-region` ... 918
`capitalize-word` ... 919
caption bar ... 795
capturing, tree-sitter node ... 1055
`car` ... 79
`car-safe` ... 79
case conversion in buffers ... 918
case conversion in Lisp ... 73
case in replacements ... 1015
`case-fold`, text property ... 390
`case-fold-search` ... 998
`case-replace` ... 998
`case-table-p` ... 75
`catch` ... 178
`catch-all` ... 594
categories of characters ... 1039
`category` (overlay property) ... 1158
`category` (text property) ... 927
category set ... 1039
category table ... 1039
`category`, in completion ... 411
category, regexp search for ... 1007
`category-docstring` ... 1040
`category-set-mnemonics` ... 1041
`category-table` ... 1040
`category-table-p` ... 1040
`cdaaar` ... 81
`cdaadr` ... 81
`cdaar` ... 81
`cdadar` ... 81
`cdaddr` ... 81
`cdadr` ... 81
`cdar` ... 81
`cddaar` ... 81
`cddadr` ... 81
`cddar` ... 81
`cdddar` ... 81
`cddddr` ... 81
`cdddr` ... 81
`cddr` ... 81
`cdr` ... 79
`cdr-safe` ... 79
`ceiling` ... 44
centering point ... 768
change hooks ... 964
change hooks for a character ... 932
change `load-path` at configure time ... 1352
`change-major-mode-after-body-hook` ... 539
`change-major-mode-hook` ... 211
changing key bindings ... 498
changing text properties ... 921
changing to another buffer ... 673
changing window size ... 705
`char-after` ... 880
`char-before` ... 880
`char-category-set` ... 1040
`char-charset` ... 976
`char-code-property-description` ... 975
`char-displayable-p` ... 1188
`char-equal` ... 60
`char-from-name` ... 971
`char-or-string-p` ... 55
`char-property-alias-alist` ... 921
`char-script-table` ... 975
`char-syntax` ... 1031
char-table length ... 102
`char-table-extra-slot` ... 120
`char-table-p` ... 120
`char-table-parent` ... 120
`char-table-range` ... 120
`char-table-subtype` ... 120
char-tables ... 119
`char-to-string` ... 66
`char-uppercase-p` ... 1162
`char-width` ... 1162
`char-width-table` ... 976
character alternative (in regexp) ... 1000
character arrays ... 54
character case ... 73
character categories ... 1039
character classes in regexp ... 1003
character code conversion ... 980
character codepoint ... 966
character codes ... 970
character event ... 441
character insertion ... 886
character printing ... 603
character properties ... 971
character set, searching ... 978
character sets ... 976
character to string ... 66
character translation tables ... 978
character width on display ... 1161
`characterp` ... 971
characters ... 54
characters for interactive codes ... 428

characters, multi-byte 966
characters, representation in
buffers and strings 966
charset 976
charset, coding systems to encode 984
`charset`, text property on buffer text 992
`charset`, text property on strings 993
`charset-after` 978
`charset-list` 976
`charset-plist` 976
`charset-priority-list` 976
`charsetp` 976
charsets supported by a coding system 985
`check-coding-system` 983
`check-coding-systems-region` 984
`check-declare-directory` 266
`check-declare-file` 266
`checkdoc` 1336
`checkdoc-current-buffer` 1336
`checkdoc-file` 1336
`checkdoc-minor-mode` 1342
`checkdoc-package-keywords` 1349
`checkdoc-package-keywords-flag` 1349
child frames 833
child process 1081
child window 695
`child-frame-border-width`, a
frame parameter 812
`child-frame-parameters`, a buffer
display action alist entry 737
choice, customization types 292
cipher, AEAD 949
cipher, symmetric 949
circular list 77
circular structure, read syntax 30
`cl` 2
`cl-call-next-method` 248
`cl-defgeneric` 245
`cl-defmethod` 246
`cl-next-method-p` 248
`cl-old-struct-compat-mode` 126
CL note—allocate more storage 1355
CL note—case of letters 15
CL note—default optional arg 234
CL note—interning existing symbol 137
CL note—lack `union`, `intersection` 91
CL note—no continuable errors 181
CL note—no `setf` functions 227
CL note—only `throw` in Emacs 178
CL note—`rplaca` vs `setcar` 88
CL note—special forms compared 150
CL note—symbol in obarrays 136
classification of file types 623
classifying events 456
`clean-mode` 528
cleanup forms 187
`clear-abbrev-table` 1069
`clear-image-cache` 1230
`clear-message-function` 1136
`clear-string` 60
`clear-this-command-keys` 438
`clear-visited-file-modtime` 681
click event 443
clickable buttons in buffers 1236
clickable text 937
clipboard 842
clipboard support (for MS-Windows) 843
`clone-indirect-buffer` 690
`clone-of`, a window parameter 781
`closepath` 1222
closure 250
closures, example of using 204
CLOS 245
`clrhash` 129
coded character set 976
codepoint, largest value 971
codes, interactive, description of 428
codespace 966
coding conventions in Emacs Lisp 1336
coding standards 1336
coding system 980
coding system for operation 989
coding system, automatically determined 986
coding system, validity check 983
coding systems for encoding a string 984
coding systems for encoding region 983
coding systems, priority 990
`coding-system-aliases` 981
`coding-system-change-eol-conversion` 983
`coding-system-change-text-conversion` 983
`coding-system-charset-list` 985
`coding-system-eol-type` 983
`coding-system-for-read` 989
`coding-system-for-write` 990
`coding-system-get` 981
`coding-system-list` 983
`coding-system-p` 983
`coding-system-priority-list` 990
`coding-system-require-warning` 990
`collapse-delayed-warnings` 1146
color names 848
`color-dark-p` 850
`color-defined-p` 849
`color-gray-p` 849
`color-name-to-rgb` 850
`color-supported-p` 849
`color-values` 849
colors on text terminals 850
column width 799
`column-0` 595
columns 912
`COM1` 1125
`combine-after-change-calls` 964
`combine-and-quote-strings` 1084
`combine-change-calls` 965
combining conditions 162

command ... 231
command descriptions ... 4
command history ... 478
command in keymap ... 495
command loop ... 424
command loop variables ... 437
command loop, recursive ... 476
`command-completion-default-include-p` ... 435
`command-debug-status` ... 344
`command-error-function` ... 182
`command-execute` ... 435
`command-history` ... 479
`command-line` ... 1266
command-line arguments ... 1266
command-line options ... 1266
`command-line-args` ... 1267
`command-line-args-left` ... 1267
`command-line-functions` ... 1267
`command-line-processed` ... 1266
`command-query` ... 478
`command-remapping` ... 505
`command-switch-alist` ... 1266
`commandp` ... 434
`commandp` example ... 405
commands, defining ... 425
commands, mode-specific ... 431
commands, specify as mode-specific ... 431
comment style ... 1029
comment syntax ... 1028
`comment-auto-fill-only-comments` ... 908
`comment-end` ... 594
`comment-end-can-be-escaped` ... 1037
`comment-start` ... 595
commentary, in a Lisp library ... 1350
comments ... 10
comments, Lisp convention for ... 1346
Common Lisp ... 1
`comp-native-version-dir` ... 304
compare tree-sitter syntax nodes ... 1054
`compare-buffer-substrings` ... 884
`compare-strings` ... 64
comparing buffer text ... 884
comparing file modification time ... 680
comparing numbers ... 42
comparing time values ... 1285
compatibility, between modules and Emacs ... 1367
compilation (Emacs Lisp) ... 317
compilation functions ... 317
compilation to native code (Emacs Lisp) ... 328
`compile-defun` ... 318
compile-time constant ... 322
compiled function ... 324
`compiled-function-p` ... 232
compiler errors ... 322
compiler macro ... 264
compiler macros, advising ... 260
compiling tree-sitter queries ... 1060
complete key ... 482
`completing-read` ... 399
`completing-read-function` ... 401
completion ... 396
completion category ... 411
completion metadata ... 411
completion styles ... 408
completion table ... 396
completion table, modifying ... 399
completion tables, combining ... 399
completion, file name ... 646
`completion-at-point` ... 412
`completion-at-point-functions` ... 413
`completion-auto-help` ... 403
`completion-boundaries` ... 398
`completion-category-overrides` ... 409
`completion-extra-properties` ... 410
`completion-ignore-case` ... 399
`completion-ignored-extensions` ... 646
`completion-in-region` ... 414
`completion-regexp-list` ... 399
`completion-styles` ... 409
`completion-styles-alist` ... 409
`completion-table-case-fold` ... 399
`completion-table-dynamic` ... 412
`completion-table-in-turn` ... 399
`completion-table-merge` ... 399
`completion-table-subvert` ... 399
`completion-table-with-cache` ... 412
`completion-table-with-predicate` ... 399
`completion-table-with-quoting` ... 399
`completion-table-with-terminator` ... 399
complex arguments ... 386
complex command ... 478
composite type, in bindat specification ... 1128
composite types (customization) ... 287
`composition` (text property) ... 934
`composition` property, and point display ... 440
`compute-motion` ... 862
computed documentation string ... 236
`concat` ... 57
concatenating bidirectional strings ... 1257
concatenating lists ... 91
concatenating strings ... 57
concurrency ... 1076
`cond` ... 160
condition name ... 186
`condition-case` ... 183
`condition-case-unless-debug` ... 183
`condition-mutex` ... 1079
`condition-name` ... 1079
`condition-notify` ... 1079
`condition-variable-p` ... 1079
`condition-wait` ... 1079
conditional evaluation ... 159
conditional selection of windows ... 721
`confirm-kill-processes` ... 1111
connection local profiles ... 218
connection local variables ... 218

connection local variables, applying 220
connection-local-criteria-alist 220
connection-local-default-application 221
connection-local-get-profile-variables ... 219
connection-local-profile-alist 219
connection-local-profile-name-for-setq ... 222
connection-local-set-profile-variables ... 218
connection-local-set-profiles 219
cons 82
cons cells 82
cons-cells-consed 1360
consing 82
consp 78
constant variables 189, 195
constrain-to-field 940
content directory, package 1308
context menus, for a major mode 531
context-menu-functions 531
continuation lines 1134
continue-process 1101
control character key constants 502
control character printing 603
control characters 13
control characters in display 1248
control characters, reading 468
control structures 158
Control-X-prefix 489
controller part, model/view/controller 1245
controlling terminal 1268
controlling-tty-p 1270
conventions for documentation strings 1342
conventions for Emacs Lisp programs 1336
conventions for Emacs programming 1340
conventions for key bindings 1339
conventions for library header comments 1348
conventions for Lisp comments 1346
conventions for writing major modes 529
conventions for writing minor modes 545
conversion of strings 65
convert buffer position to file byte 967
convert file byte to buffer position 967
convert sequence to another type 114
convert-standard-filename 647
converting file names from/to
  MS-Windows syntax 636
converting numbers 43
coordinate, relative to frame 772
Coordinated Universal Time 1275
coordinates-in-window-p 773
copy-abbrev-table 1069
copy-alist 97
copy-category-table 1040
copy-directory 651
copy-file 633
copy-hash-table 131
copy-keymap 487
copy-marker 873
copy-overlay 1155
copy-region-as-kill 893
copy-sequence 103
copy-syntax-table 1030
copy-tree 84
copy_string_contents 1373
copying alists 97
copying bidirectional text,
  preserve visual order 1259
copying files 631
copying lists 83
copying sequences 103
copying strings 57
copying vectors 118
copysign 41
cos 51
count-lines 860
count-loop 5
count-screen-lines 861
count-words 860
counting columns 912
counting set bits 51
coverage testing 370
coverage testing (Edebug) 357
create subprocess 1081
create-file-buffer 613
create-fontset-from-fontset-spec 1186
create-image 1225
create-lockfiles 620
creating buffers 686
creating hash tables 127
creating keymaps 484
creating markers 872
creating strings 55
creating tree-sitter parsers 1047
creating, copying and deleting directories 650
cryptographic hash 947, 949
ctl-arrow 1248
ctl-x-4-map 489
ctl-x-5-map 489
ctl-x-map 489
ctl-x-r-map 1400
curly quotes 602, 1344
curly quotes, in formatted messages 66
current binding 190
current buffer 673
current buffer mark 876
current buffer point and mark (Edebug) 358
current buffer position 855
current command 437
current stack frame 339
current-active-maps 491
current-bidi-paragraph-direction 1257
current-buffer 673
current-case-table 75
current-column 912
current-cpu-time 1277
current-fill-column 905
current-frame-configuration 833

current-global-map ... 492
current-idle-time ... 1290
current-indentation ... 913
current-input-method ... 994
current-input-mode ... 1291
current-justification ... 904
current-kill ... 895
current-left-margin ... 905
current-local-map ... 492
current-message ... 1138
current-minibuffer-command ... 438
current-minor-mode-maps ... 493
current-prefix-arg ... 475
current-thread ... 1077
current-time ... 1276
current-time-list ... 1276
current-time-string ... 1276
current-time-zone ... 1277
current-window-configuration ... 777
current-word ... 883
currying ... 240
cursor ... 761
cursor (text property) ... 931
cursor position for display properties and overlays ... 931
cursor, and frame parameters ... 819
cursor, fringe ... 1196
cursor-color, a frame parameter ... 822
cursor-face (text property) ... 928
cursor-face-highlight-mode ... 928
cursor-face-highlight-nonselected-window ... 928
cursor-in-echo-area ... 1142
cursor-in-non-selected-windows ... 819
cursor-intangible (text property) ... 930
cursor-intangible-mode ... 930
cursor-sensor-functions (text property) ... 933
cursor-sensor-inhibit ... 930
cursor-sensor-mode ... 933
cursor-type ... 819
cursor-type, a frame parameter ... 819
curved quotes ... 602, 1344
curved quotes, in formatted messages ... 66
curveto ... 1222
custom '%'-sequence in format ... 70
custom format string ... 70
custom themes ... 296
custom-add-frequent-value ... 284
custom-group property ... 281
custom-initialize-delay ... 1353
custom-known-themes ... 297
custom-reevaluate-setting ... 285
custom-set-faces ... 295
custom-set-variables ... 295
custom-theme-p ... 297
custom-theme-set-faces ... 297
custom-theme-set-variables ... 296
custom-unlispify-remove-prefixes ... 281
custom-variable-history ... 395
custom-variable-p ... 285
customizable variables, how to define ... 281
customization groups, defining ... 280
customization item ... 278
customization keywords ... 278
customization types ... 285
customization types, define new ... 294
customize-package-emacs-version-alist ... 280
customized-face (face symbol property) ... 1174
cycle-sort-function, in completion ... 412
cyclic ordering of windows ... 719
cygwin-convert-file-name-from-windows ... 636
cygwin-convert-file-name-to-windows ... 636

D

dangling symlinks, testing for existence ... 621
data layout specification ... 1128
data type ... 8
data-directory ... 606
database access, SQLite ... 952
database object ... 952
datagrams ... 1118
date-days-in-month ... 1286
date-leap-year-p ... 1286
date-ordinal-to-time ... 1286
date-to-time ... 1281
days-to-time ... 1286
deactivate-mark ... 877, 878
deactivate-mark-hook ... 878
debug ... 341
debug-allow-recursive-debug ... 336, 341
debug-ignored-errors ... 335
debug-on-entry ... 337
debug-on-error ... 335
debug-on-error use ... 182
debug-on-event ... 336
debug-on-message ... 336
debug-on-next-call ... 344
debug-on-quit ... 337
debug-on-signal ... 335
debug-on-variable-change ... 338
debugger ... 343
debugger command list ... 340
debugger for Emacs Lisp ... 334
debugger, explicit entry ... 338
debugger-bury-or-kill ... 339
debugger-stack-frame-as-list ... 344
debugging changes to variables ... 338
debugging errors ... 334
debugging font-lock ... 573
debugging invalid Lisp syntax ... 368
debugging lisp programs ... 334
debugging redisplay errors ... 336
debugging specific functions ... 337
declare ... 263
declare-function ... 265, 266

declaring functions ... 265
decode process output ... 1106
decode-char ... 977
decode-coding-inserted-region ... 993
decode-coding-region ... 992
decode-coding-string ... 992
decode-time ... 1279
decoding file formats ... 656
decoding in coding systems ... 991
decrement field of register ... 16
dedicated window ... 751
dedicated, a buffer display action alist entry ... 736
def-edebug-elem-spec ... 362
def-edebug-spec ... 360
defadvice ... 258
defalias ... 238
defalias-fset-function property ... 238
default argument string ... 428
default character height ... 799
default character size ... 799
default character width ... 799
default coding system ... 986
default coding system,
  functions to determine ... 988
default filter function of a process ... 1104
default font ... 799
default height of character ... 799
default init file ... 1264
default key binding ... 483
default sentinel function of a process ... 1109
default value ... 211
default value of char-table ... 119
default width of character ... 799
default-boundp ... 212
default-directory ... 642
default-file-modes ... 635
default-font-height ... 1193
default-font-width ... 1193
default-frame-alist ... 806
default-input-method ... 994
default-justification ... 904
default-minibuffer-frame ... 827
default-process-coding-system ... 988
default-text-properties ... 921
default-toplevel-value ... 213
default-value ... 212
default.el ... 1261
defconst ... 195
defcustom ... 281
deferred evaluation ... 155
defface ... 1172
defgroup ... 281
defimage ... 1225
define customization group ... 280
define customization options ... 281
define hash comparisons ... 130
define image ... 1225
define new bindat type forms ... 1132
define new customization types ... 294
define-abbrev ... 1070
define-abbrev-table ... 1070
define-advice ... 256
define-alternatives ... 433
define-button-type ... 1238
define-category ... 1039
define-derived-mode ... 535
define-error ... 186
define-fringe-bitmap ... 1198
define-generic-mode ... 542
define-globalized-minor-mode ... 550
define-hash-table-test ... 130
define-icon ... 1231
define-inline ... 262
define-key ... 502
define-key-after ... 503
define-keymap ... 485
define-minor-mode ... 547
define-multisession-variable ... 228
define-obsolete-face-alias ... 1182
define-obsolete-function-alias ... 261
define-obsolete-variable-alias ... 223
define-package ... 1311
define-prefix-command ... 490
define-short-documentation-group ... 607
defined-colors ... 849
defining a function ... 237
defining abbrevs ... 1070
defining commands ... 425
defining customization variables in C ... 1364
defining faces ... 1172
defining functions dynamically ... 238
defining Lisp variables in C ... 1364
defining macros ... 271
defining menus ... 511
defining tokens, SMIE ... 586
defining-kbd-macro ... 480
definition-prefixes ... 308
definitions of symbols ... 134
defmacro ... 271
defsubr, Lisp symbol for a primitive ... 1363
defsubst ... 261
deftheme ... 296
defun ... 237
defun-prompt-regexp ... 864
DEFUN, C macro to define Lisp primitives ... 1361
defvar ... 194
defvar-keymap ... 486
defvar-local ... 209
DEFVAR_INT, DEFVAR_LISP,
  DEFVAR_BOOL, DEFSYM ... 1364
defvaralias ... 222
delay-mode-hooks ... 539
delay-warning ... 1146
delayed warnings ... 1146
delayed-warnings-hook ... 1146, 1404
delayed-warnings-list ... 1146

`delete` ... 93
`delete-and-extract-region` ... 888
`delete-auto-save-file-if-necessary` ... 668
`delete-auto-save-files` ... 669
`delete-backward-char` ... 888
`delete-before`, a frame parameter ... 815
`delete-blank-lines` ... 891
`delete-by-moving-to-trash` ... 633, 651
`delete-char` ... 888
`delete-directory` ... 651
`delete-dups` ... 94
`delete-exited-processes` ... 1095
`delete-field` ... 940
`delete-file` ... 633
`delete-frame` ... 824
`delete-frame` event ... 451
`delete-frame-functions` ... 824
`delete-horizontal-space` ... 889
`delete-indentation` ... 890
`delete-minibuffer-contents` ... 421
`delete-old-versions` ... 664
`delete-other-frames` ... 825
`delete-other-windows` ... 713
`delete-other-windows`, a window parameter ... 780
`delete-overlay` ... 1155
`delete-process` ... 1095
`delete-region` ... 888
`delete-selection`, symbol property ... 879
`delete-selection-helper` ... 879
`delete-selection-pre-hook` ... 879
`delete-terminal` ... 790
`delete-terminal-functions` ... 790
`delete-to-left-margin` ... 906
`delete-trailing-whitespace` ... 891
`delete-window` ... 713
`delete-window`, a window parameter ... 780
`delete-window-choose-selected` ... 713
`delete-windows-on` ... 714
deleting files ... 631
deleting frames ... 824
deleting list elements ... 92
deleting previous char ... 888
deleting processes ... 1094
deleting text vs killing ... 888
deleting whitespace ... 889
deleting windows ... 713
`delq` ... 92
dependencies ... 1308
derived mode ... 535
`derived-mode-p` ... 537
describe characters and events ... 603
`describe-bindings` ... 510
`describe-buffer-case-table` ... 76
`describe-categories` ... 1041
`describe-current-display-table` ... 1249
`describe-display-table` ... 1249
`describe-mode` ... 535
`describe-prefix-bindings` ... 605
`describe-syntax` ... 1031
description for interactive codes ... 428
description format ... 4
deserializing ... 1128
desktop notifications ... 1296
desktop save mode ... 595
`desktop-buffer-mode-handlers` ... 596
`desktop-save-buffer` ... 596
`destroy-fringe-bitmap` ... 1198
destructive list operations ... 87
destructuring with pcase patterns ... 172
`detect-coding-region` ... 984
`detect-coding-string` ... 984
deterministic build ... 1352
device names ... 439
`device-class` ... 440
diagrams, boxed, for lists ... 16
dialog boxes ... 841
`digit-argument` ... 476
`ding` ... 1252
`dir-locals-class-alist` ... 218
`dir-locals-directory-cache` ... 218
`dir-locals-file` ... 217
`dir-locals-set-class-variables` ... 217
`dir-locals-set-directory-class` ... 218
direct save protocol ... 845
`direction`, a buffer display action alist entry ... 737
directional overrides ... 1258
directory file name ... 640
directory local variables ... 217
directory name ... 640
directory part (of file name) ... 637
`directory-abbrev-alist` ... 641
`directory-empty-p` ... 648
`directory-file-name` ... 640
`directory-files` ... 648
`directory-files-and-attributes` ... 649
`directory-files-no-dot-files-regexp` ... 649
`directory-files-recursively` ... 648
`directory-name-p` ... 640
directory-oriented functions ... 648
`dired-kept-versions` ... 664
disable asynchronous native compilation ... 332
`disable-command` ... 478
`disable-point-adjustment` ... 441
`disable-theme` ... 298
`disabled` ... 478
disabled command ... 478
`disabled-command-function` ... 478
disabling multibyte ... 968
disabling undo ... 901
`disassemble` ... 325
disassembled byte-code ... 325
`discard-input` ... 470
discarding input ... 470
dispatch of methods for generic function ... 248
`display` (overlay property) ... 1158
`display` (text property) ... 1203

display action ... 726
display area ... 797
display feature testing ... 852
display margins ... 1209
display message in echo area ... 1135
display name on X ... 790
display origin ... 797
display properties, and bidi reordering of text ... 1256
display property ... 1203
`display` property, and point display ... 440
display property, unsafe evaluation ... 1204
display specification ... 1203
display table ... 1248
`display`, a frame parameter ... 807
display, abstract ... 1241
display, arbitrary objects ... 1241
`display-backing-store` ... 854
`display-buffer` ... 727
`display-buffer-alist` ... 728
`display-buffer-at-bottom` ... 731
`display-buffer-base-action` ... 728
`display-buffer-below-selected` ... 731
`display-buffer-fallback-action` ... 728
`display-buffer-full-frame` ... 732
`display-buffer-in-atom-window` ... 760
`display-buffer-in-child-frame` ... 732
`display-buffer-in-direction` ... 730
`display-buffer-in-previous-window` ... 729
`display-buffer-in-side-window` ... 754
`display-buffer-no-window` ... 732
`display-buffer-overriding-action` ... 728
`display-buffer-pop-up-frame` ... 732
`display-buffer-pop-up-window` ... 729
`display-buffer-reuse-mode-window` ... 729
`display-buffer-reuse-window` ... 728
`display-buffer-same-window` ... 728
`display-buffer-use-least-recent-window` ... 730
`display-buffer-use-some-frame` ... 732
`display-buffer-use-some-window` ... 730
`display-color-cells` ... 854
`display-color-p` ... 852
`display-completion-list` ... 402
`display-delayed-warnings` ... 1146
`display-graphic-p` ... 852
`display-grayscale-p` ... 852
`display-images-p` ... 853
`display-message-or-buffer` ... 1138
`display-mm-dimensions-alist` ... 854
`display-mm-height` ... 853
`display-mm-width` ... 853
`display-monitor-attributes-list` ... 791
`display-monitors-changed-functions` ... 793
`display-mouse-p` ... 852
`display-pixel-height` ... 853
`display-pixel-width` ... 853
`display-planes` ... 854
`display-popup-menus-p` ... 852
`display-save-under` ... 854
`display-screens` ... 853
`display-selections-p` ... 853
`display-sort-function`, in completion ... 412
display-start position ... 762
`display-supports-face-attributes-p` ... 852
`display-table-slot` ... 1249
`display-type`, a frame parameter ... 807
`display-visual-class` ... 854
`display-warning` ... 1143
displaying a buffer ... 726
displaying faces ... 1179
displays, multiple ... 789
distance between strings ... 64
distinguish interactive calls ... 436
`dlet` ... 192
`dnd-begin-drag-files` ... 847
`dnd-begin-file-drag` ... 846
`dnd-begin-text-drag` ... 846
`dnd-direct-save` ... 847
`dnd-protocol-alist` ... 844
`do-auto-save` ... 668
`doc`, customization keyword ... 293
`doc-directory` ... 600
`DOC` (documentation) file ... 597
Document Object Model ... 956
`documentation` ... 598
documentation conventions ... 597
documentation for major mode ... 535
documentation groups ... 607
documentation notation ... 3
documentation string of function ... 235
documentation strings ... 597
documentation strings, conventions and tips ... 1342
documentation, keys in ... 600
`documentation-property` ... 598
`dolist` ... 175
`dolist-with-progress-reporter` ... 1140
`dom-node` ... 956
DOM ... 956
`dotimes` ... 175
`dotimes-with-progress-reporter` ... 1140
dotted list ... 77
dotted lists (Edebug) ... 363
dotted pair notation ... 18
double-click events ... 447
`double-click-fuzz` ... 448
`double-click-time` ... 448
double-quote in strings ... 20
`down-list` ... 863
`downcase` ... 73
`downcase-region` ... 918
`downcase-word` ... 919
downcasing in `lookup-key` ... 463
`download-callback` xwidget events ... 451
`download-started` xwidget events ... 451
drag and drop ... 844

drag and drop protocols, X 848
drag and drop, other formats 844
drag and drop, X 844
drag event 446
`drag-internal-border`, a frame parameter 816
`drag-n-drop` event 454
`drag-with-header-line`, a frame parameter 816
`drag-with-mode-line`, a frame parameter 816
`drag-with-tab-line`, a frame parameter 816
dribble file 1292
drop target, in drag-and-drop operations 846
dump file 1351
`dump-emacs` 1353
`dump-emacs-portable` 1353
dumping Emacs 1351
dynamic binding 201
dynamic binding, temporarily 192
dynamic extent 201
dynamic let-binding 192
dynamic libraries 1304
dynamic loading of documentation 320
dynamic loading of functions 320
dynamic modules 315
dynamic modules, writing 1366
dynamic scope 201
`dynamic-library-alist` 1304

E

eager macro expansion 300
early init file 1263
`early-init.el` 1263
`easy-menu-define` 522
`easy-mmode-define-minor-mode` 549
echo area 1135
echo area customization 1142
`echo-area-clear-hook` 1142
`echo-keystrokes` 1142
`edebug` 352
Edebug debugging facility 345
Edebug execution modes 348
edebug overwrites buffer point position 358
Edebug specification list 361
edebug, failure to instrument 347
`edebug-after-instrumentation-function` 368
`edebug-all-defs` 365
`edebug-all-forms` 366
`edebug-backtrace-hide-instrumentation` 350
`edebug-backtrace-show-instrumentation` 350
`edebug-behavior-alist` 368
`edebug-continue-kbd-macro` 367
`edebug-defun` 347
`edebug-display-freq-count` 357
`edebug-eval-macro-args` 360, 366
`edebug-eval-top-level-form` 347
`edebug-global-break-condition` 368
`edebug-initial-mode` 366
`edebug-max-depth` 358
`edebug-new-definition-function` 368
`edebug-on-error` 367
`edebug-on-quit` 367
`edebug-print-circle` 356, 367
`edebug-print-length` 355, 367
`edebug-print-level` 356, 367
`edebug-print-trace-after` 356
`edebug-print-trace-before` 356
`edebug-remove-instrumentation` 347
`edebug-save-displayed-buffer-points` 366
`edebug-save-windows` 366
`edebug-set-global-break-condition` 352
`edebug-set-initial-mode` 348
`edebug-setup-hook` 365
`edebug-sit-for-seconds` 349, 368
`edebug-sit-on-break` 368
`edebug-temp-display-freq-count` 357
`edebug-test-coverage` 367
`edebug-trace` 356, 367
`edebug-tracing` 356
`edebug-unwrap-results` 367
edge detection, images 1214
edit distance between strings 64
`edit-and-eval-command` 393
editing types 26
editor command loop 424
`eight-bit`, a charset 976
`electric-future-map` 6
element (of list) 77
elements of sequences 103
`elliptical-arc` 1224
`elp.el` 371
`elt` 103
Emacs event standard notation 603
Emacs process run time 1285
`emacs`, a charset 976
`emacs-build-number` 7
`emacs-build-time` 6
`emacs-init-time` 1285
`emacs-internal` coding system 981
`emacs-lisp-docstring-fill-column` 1343
`emacs-lisp-native-compile` 330
`emacs-lisp-native-compile-and-load` 330
`emacs-major-version` 6
`emacs-minor-version` 7
`emacs-pid` 1273
`emacs-repository-branch` 7
`emacs-repository-version` 7
`emacs-save-session-functions` 1296
`emacs-session-restore` 1296
`emacs-startup-hook` 1264
`emacs-uptime` 1285
`emacs-version` 6
`emacs_finalizer` 1371
`emacs_funcall_exit`, enumeration 1381
`emacs_funcall_exit_return` 1381
`emacs_funcall_exit_signal` 1381
`emacs_funcall_exit_throw` 1381

emacs_function ... 1368
emacs_module_init ... 315, 1366
emacs_value data type ... 1371
emacs_variadic_function ... 1369
EMACS_LIMB_MAX ... 1372
emacsclient, getting a backtrace ... 335
EMACSLOADPATH environment variable ... 303
embedded language, tree-sitter ... 1062
embedded widgets ... 1233
empty file name ... 640
empty file name, and file-exists-p ... 621
empty file names, and expand-file-name ... 642
empty lines, indicating ... 1195
empty list ... 17
empty overlay ... 1154
empty region ... 879
emulation-mode-map-alists ... 494
enable-command ... 478
enable-connection-local-variables ... 222
enable-dir-local-variables ... 218
enable-local-eval ... 216
enable-local-variables ... 214
enable-multibyte-characters ... 966, 968
enable-recursive-minibuffers ... 422
enable-theme ... 298
encapsulation, ewoc ... 1241
encode-char ... 977
encode-coding-region ... 991
encode-coding-string ... 992
encode-time ... 1280
encoding file formats ... 656
encoding in coding systems ... 991
encrypted network connections ... 1115
end of line in regexp ... 1002
end-of-buffer ... 858
end-of-defun ... 864
end-of-defun-function ... 864
end-of-file ... 375
end-of-line ... 859
end-of-line conversion ... 980
end-session event ... 455
endianness, in bindat specification ... 1128
ensure-empty-lines ... 887
ensure-list ... 84
environment ... 145
environment variable access ... 1271
environment variables, subprocesses ... 1082
eobp ... 881
eol conversion of coding system ... 983
eol type of coding system ... 983
EOL conversion ... 980
eolp ... 881
epoch ... 1275
eq ... 34, 1378
eql ... 42
equal ... 35
equal-including-properties ... 36
equality ... 34
erase-buffer ... 888
error ... 180
error cleanup ... 187
error debugging ... 334
error description ... 184
error display ... 1135
error handler ... 182
error in debug ... 342
error message notation ... 3
error name ... 186
error symbol ... 186
error-conditions ... 186
error-message-string ... 184
errors ... 179
esc-map ... 489
ESC-prefix ... 489
escape (ASCII character) ... 12
escape characters ... 381
escape characters in printing ... 378
escape sequence ... 12
escaping apostrophe and grave accent in doc strings ... 1344
ESC ... 498
eval ... 153
eval during compilation ... 321
eval, and debugging ... 344
eval-and-compile ... 321
eval-buffer ... 154
eval-buffer (Edebug) ... 347
eval-defun (Edebug) ... 347
eval-defun, and defcustom forms ... 282
eval-defun, and defface forms ... 1173
eval-defun, and defvar forms ... 195
eval-defun, and explicit entry to debugger ... 338
eval-expression (Edebug) ... 347
eval-expression, and lexical-binding ... 205
eval-expression-debug-on-error ... 335
eval-expression-print-length ... 382
eval-expression-print-level ... 382
eval-last-sexp, and defface forms ... 1173
eval-last-sexp, and defvar forms ... 195
eval-minibuffer ... 392
eval-region ... 153
eval-region (Edebug) ... 347
eval-when-compile ... 322
evaluated expression argument ... 430
evaluation ... 145
evaluation list group ... 354
evaluation notation ... 3
evaluation of buffer contents ... 154
evaluation of special forms ... 149
evaporate (overlay property) ... 1160
even-window-sizes ... 739
event printing ... 603
event translation ... 467
event type ... 456
event, reading only one ... 463
event-basic-type ... 457

`event-click-count` ... 448
`event-convert-list` ... 504
`event-end` ... 458
`event-modifiers` ... 456
`event-start` ... 457
`eventp` ... 441
events ... 441
ewoc ... 1241
`ewoc-buffer` ... 1242
`ewoc-collect` ... 1243
`ewoc-create` ... 1242
`ewoc-data` ... 1243
`ewoc-delete` ... 1243
`ewoc-enter-after` ... 1242
`ewoc-enter-before` ... 1242
`ewoc-enter-first` ... 1242
`ewoc-enter-last` ... 1242
`ewoc-filter` ... 1243
`ewoc-get-hf` ... 1242
`ewoc-goto-next` ... 1243
`ewoc-goto-node` ... 1243
`ewoc-goto-prev` ... 1243
`ewoc-invalidate` ... 1243
`ewoc-locate` ... 1243
`ewoc-location` ... 1243
`ewoc-map` ... 1244
`ewoc-next` ... 1242
`ewoc-nth` ... 1243
`ewoc-prev` ... 1242
`ewoc-refresh` ... 1243
`ewoc-set-data` ... 1243
`ewoc-set-hf` ... 1242
examining text properties ... 920
examining the `interactive` form ... 427
examining windows ... 722
examples of using `interactive` ... 431
excess close parentheses ... 369
excess open parentheses ... 369
excursion ... 866
`exec-directory` ... 1082
`exec-path` ... 1082, 1083
`exec-suffixes` ... 1082
`executable-find` ... 631
execute program ... 1081
execute with prefix argument ... 435
`execute-extended-command` ... 435
`execute-extended-command-for-buffer` ... 435
`execute-kbd-macro` ... 479
`executing-kbd-macro` ... 480
execution order of buffer display action functions ... 741
execution speed ... 1341
`exit` ... 476
exit recursive editing ... 476
`exit-minibuffer` ... 418
`exit-recursive-edit` ... 477
exiting Emacs ... 1267
`exp` ... 51
`expand-abbrev` ... 1072
`expand-file-name` ... 641
expanding abbrevs ... 1072
expansion of file names ... 641
expansion of macros ... 269
explicit selective display ... 1149
`explicit-name`, a frame parameter ... 807
explore tree-sitter syntax tree ... 1044
expression ... 145
`expt` ... 52
extended file attributes ... 629
extended menu item ... 512
`extended-command-history` ... 395
extent ... 201
external border ... 794
external menu bar ... 795
external tool bar ... 795
`external-debugging-output` ... 378
extra node, tree-sitter ... 1054
extra slots of char-table ... 119
`extra-keyboard-modifiers` ... 467
`extract_big_integer` ... 1372
`extract_float` ... 1373
`extract_integer` ... 1372
`extract_time` ... 1373

F

`face` (button property) ... 1237
face (non-removability of) ... 1172
`face` (overlay property) ... 1158
`face` (text property) ... 927
face alias ... 1182
face attributes ... 1168
face attributes, access and modification ... 1175
face codes of text ... 927
face merging ... 1179
face name ... 1168
face number ... 1181
face property of face symbols ... 1181
face remapping ... 1179
face spec ... 1172
`face-all-attributes` ... 1176
`face-attribute` ... 1175
`face-attribute-relative-p` ... 1175
`face-background` ... 1178
`face-bold-p` ... 1178
`face-defface-spec` (face symbol property) ... 1174
`face-differs-from-default-p` ... 1182
`face-documentation` ... 598, 1181
`face-documentation` (face symbol property) ... 1174
`face-equal` ... 1182
`face-extend-p` ... 1178
`face-filters-always-match` ... 934
`face-font` ... 1178
`face-font-family-alternatives` ... 1184
`face-font-registry-alternatives` ... 1185

face-font-rescale-alist ... 1185
face-font-selection-order ... 1184
face-foreground ... 1178
face-id ... 1181
face-inverse-video-p ... 1178
face-italic-p ... 1178
face-list ... 1181
face-name-history ... 395
face-remap-add-relative ... 1180
face-remap-remove-relative ... 1181
face-remap-reset-base ... 1181
face-remap-set-base ... 1181
face-remapping-alist ... 1180
face-spec-set ... 1174
face-stipple ... 1178
face-underline-p ... 1178
facep ... 1168
faces ... 1168
faces for font lock ... 574
faces, automatic choice ... 1182
false ... 2
fboundp ... 249
fceiling ... 48
feature-unload-function ... 314
featurep ... 311
features ... 310, 312
fetch-bytecode ... 321
ffloor ... 48
field (overlay property) ... 1159
field (text property) ... 930
field name, tree-sitter ... 1044
field numbers in format spec ... 68
field width ... 69
field-beginning ... 940
field-end ... 940
field-is ... 593
field-string ... 940
field-string-no-properties ... 940
fields ... 939
fifo data structure ... 124
file accessibility ... 621
file age ... 626
file attributes ... 626
file classification ... 623
file contents, and default coding system ... 986
file format conversion ... 656
file hard link ... 631
file local variables ... 213, 1305
file locks ... 619
file mode specification error ... 533
file modes ... 623
file modes and MS-DOS ... 623
file modes, setting ... 634
file modification time ... 626
file name abbreviations ... 641
file name completion subroutines ... 646
file name handler ... 651
file name of buffer ... 677
file name of directory ... 640
file name, and default coding system ... 987
file names ... 636
file names in directory ... 648
file names, trailing whitespace ... 621
file notifications ... 1301
file open error ... 613
file permissions ... 623
file permissions, setting ... 634
file with multiple names ... 631
file, ancestor directory of ... 649
file, information about ... 621
file-accessible-directory-p ... 622
file-acl ... 629
file-already-exists ... 633
file-attributes ... 627
file-backup-file-names ... 666
file-chase-links ... 625
file-coding-system-alist ... 987
file-directory-p ... 624
file-equal-p ... 626
file-error ... 301
file-executable-p ... 622
file-exists-p ... 621
file-expand-wildcards ... 649
file-extended-attributes ... 630
file-has-changed-p ... 627
file-in-directory-p ... 649
file-local-copy ... 654
file-local-name ... 655
file-local-variables-alist ... 215
file-locked ... 620
file-locked-p ... 620
file-modes ... 623
file-modes-number-to-symbolic ... 635
file-modes-symbolic-to-number ... 635
file-name encoding, MS-Windows ... 982
file-name-absolute-p ... 639
file-name-all-completions ... 646
file-name-as-directory ... 640
file-name-base ... 638
file-name-case-insensitive-p ... 626
file-name-coding-system ... 982
file-name-completion ... 646
file-name-concat ... 640
file-name-directory ... 637
file-name-extension ... 638
file-name-handler-alist ... 651
file-name-history ... 395
file-name-nondirectory ... 637
file-name-parent-directory ... 641
file-name-quote ... 643
file-name-quoted-p ... 643
file-name-sans-extension ... 638
file-name-sans-versions ... 637
file-name-split ... 639
file-name-unquote ... 643
file-name-with-extension ... 638

file-newer-than-file-p ... 626
file-newest-backup ... 666
file-nlinks ... 629
file-notify-add-watch ... 1301
file-notify-rm-all-watches ... 1303
file-notify-rm-watch ... 1303
file-notify-valid-p ... 1303
file-ownership-preserved-p ... 622
file-precious-flag ... 616
file-readable-p ... 622
file-regular-p ... 625
file-relative-name ... 639
file-remote-p ... 655
file-selinux-context ... 630
file-supersession ... 681
file-symlink-p ... 623
file-truename ... 625
file-writable-p ... 622
filepos-to-bufferpos ... 967
fill-column ... 905
fill-context-prefix ... 906
fill-forward-paragraph-function ... 904
fill-individual-paragraphs ... 903
fill-individual-varying-indent ... 903
fill-nobreak-predicate ... 906
fill-paragraph ... 902
fill-paragraph-function ... 904
fill-prefix ... 905
fill-region ... 902
fill-region-as-paragraph ... 903
fill-separate-heterogeneous-words-with-space ... 904
fillarray ... 117
filling text ... 901
filling, automatic ... 907
filter function ... 1104
filter multibyte flag, of process ... 1107
filter-buffer-substring ... 882
filter-buffer-substring-function ... 882
filter-buffer-substring-functions ... 883
filtered-frame-list ... 826
filtering killed text ... 892
filtering sequences ... 109
find file in path ... 630
find library ... 302
find-auto-coding ... 988
find-backup-file-name ... 665
find-buffer-visiting ... 678
find-charset-region ... 978
find-charset-string ... 978
find-coding-systems-for-charsets ... 984
find-coding-systems-region ... 983
find-coding-systems-string ... 984
find-file ... 611
find-file-hook ... 612
find-file-literally ... 611, 613
find-file-name-handler ... 654
find-file-noselect ... 611
find-file-not-found-functions ... 612
find-file-other-window ... 612
find-file-read-only ... 612
find-file-wildcards ... 612
find-font ... 1190
find-image ... 1226
find-operation-coding-system ... 989
find-word-boundary-function-table ... 857
finding files ... 610
finding windows ... 721
first-change-hook ... 965
first-sibling ... 594
fit-frame-to-buffer ... 707
fit-frame-to-buffer-margins ... 708
fit-frame-to-buffer-margins, a frame parameter ... 812
fit-frame-to-buffer-sizes ... 708
fit-frame-to-buffer-sizes, a frame parameter ... 812
fit-window-to-buffer ... 706
fit-window-to-buffer-horizontally ... 707
fixed-size window ... 704
fixnump ... 41
fixup-whitespace ... 890
flags in format specifications ... 68
flatten-tree ... 84
float ... 43
float-e ... 52
float-output-format ... 383
float-pi ... 52
float-time ... 1277
floating-point functions ... 51
floatp ... 41
floats-consed ... 1360
floor ... 44
flowcontrol, in serial connections ... 1127
flush-standard-output ... 380
flushing input ... 470
fmakunbound ... 249
fn in function's documentation string ... 307
focus event ... 450
focus-follows-mouse ... 830
focus-in-hook ... 1404
focus-out-hook ... 1404
follow links ... 937
follow-link (button property) ... 1237
follow-link (text or overlay property) ... 938
following-char ... 880
font and color, frame parameters ... 820
font backend ... 1190
font entity ... 1190
font information for layout ... 1193
font lock faces ... 574
Font Lock mode ... 564
font lookup ... 1185
font object ... 1188
font property ... 1188
font registry ... 1189

font selection 1184
font spec 1189
font, a frame parameter 821
font-at 1189
font-backend, a frame parameter 820
font-face-attributes 1191
font-family-list 1172
font-get 1191
font-info 1191
font-lock-add-keywords 570
font-lock-bracket-face 576
font-lock-builtin-face 575
font-lock-comment-delimiter-face 575
font-lock-comment-face 575
font-lock-constant-face 575
font-lock-debug-fontify 565
font-lock-defaults 566
font-lock-delimiter-face 576
font-lock-doc-face 575
font-lock-doc-markup-face 576
font-lock-ensure &optional beg end 565
font-lock-ensure-function 573
font-lock-escape-face 576
font-lock-extend-after-change-region-function 579
font-lock-extra-managed-props 572
font-lock-face (text property) 928
font-lock-flush &optional beg end 565
font-lock-flush-function 573
font-lock-fontify-buffer 565
font-lock-fontify-buffer-function 572
font-lock-fontify-region beg end &optional loudly 565
font-lock-fontify-region-function 573
font-lock-function-call-face 575
font-lock-function-name-face 575
font-lock-ignore 571
font-lock-keyword-face 575
font-lock-keywords 567
font-lock-keywords-case-fold-search 570
font-lock-keywords-only 577
font-lock-mark-block-function 572
font-lock-misc-punctuation-face 576
font-lock-multiline 579
font-lock-negation-char-face 576
font-lock-number-face 576
font-lock-operator-face 576
font-lock-preprocessor-face 575
font-lock-property-name-face 576
font-lock-property-use-face 576
font-lock-punctuation-face 576
font-lock-remove-keywords 570
font-lock-string-face 575
font-lock-syntactic-face-function 577
font-lock-syntax-table 577
font-lock-type-face 575
font-lock-unfontify-buffer 565
font-lock-unfontify-buffer-function 573
font-lock-unfontify-region beg end 565
font-lock-unfontify-region-function 573
font-lock-variable-name-face 575
font-lock-variable-use-face 575
font-lock-warning-face 575
font-put 1190
font-spec 1189
font-xlfd-name 1191
fontification-functions 1182
fontifications with tree-sitter, overview 580
fontified (text property) 928
fontp 1188
fontset 1186
foo 4
for 272
force coding system for operation 989
force entry to debugger 338
force-mode-line-update 551
force-window-update 1134
forcing redisplay 1133
foreground-color, a frame parameter 821
form 145
form, self-evaluating 146
format 66
format definition 657
format of gnutls cryptography inputs 950
format of keymaps 482
format specification 67
format, customization keyword 292
format-alist 657
format-find-file 658
format-insert-file 658
format-message 66
format-mode-line 561
format-network-address 1124
format-prompt 391
format-seconds 1284
format-spec 70
format-time-string 1281
format-write-file 658
formatting numbers for rereading later 69
formatting strings 66
formatting time values 1281
formfeed 12
forms for cleanup 187
forms for control structures 158
forms for handling errors 182
forms for nonlocal exits 177
forms for sequential execution 158
forms, backquote 151
forms, conditional 159
forms, function call 148
forms, iteration 174
forms, list 147
forms, macro call 149
forms, quote 151
forms, special 149
forms, symbol 146

`forward-button` ... 1240
`forward-char` ... 856
`forward-comment` ... 1034
`forward-line` ... 859
`forward-list` ... 863
`forward-sexp` ... 864
`forward-to-indentation` ... 918
`forward-word` ... 857
`forward-word-strictly` ... 858
frame ... 787
frame configuration ... 833
frame creation ... 788
frame geometry ... 793
frame height ratio ... 810
frame interaction parameters ... 814
frame layout ... 793
frame layout parameters ... 812
frame parameters ... 804
frame parameters for windowed displays ... 807
frame position ... 793, 800, 807
frame position changes, a hook ... 800
frame size ... 793, 801
frame stacking order ... 832
frame title ... 823
frame visibility ... 831
frame width ratio ... 810
frame without a minibuffer ... 826
frame Z-order ... 832
frame, which buffers to display ... 814
`frame-alpha-lower-limit` ... 821
`frame-ancestor-p` ... 837
`frame-auto-hide-function` ... 754
`frame-char-height` ... 799
`frame-char-width` ... 799
`frame-current-scroll-bars` ... 1200
`frame-edges` ... 798
`frame-first-window` ... 696
`frame-focus-state` ... 829
`frame-geometry` ... 797
`frame-height` ... 801
`frame-inherited-parameters` ... 789
`frame-inhibit-implied-resize` ... 804
`frame-inner-height` ... 801
`frame-inner-width` ... 801
`frame-list` ... 825
`frame-list-z-order` ... 825
`frame-live-p` ... 825
`frame-monitor-attributes` ... 792
`frame-native-height` ... 801
`frame-native-width` ... 801
`frame-old-selected-window` ... 786
`frame-outer-height` ... 801
`frame-outer-width` ... 801
`frame-parameter` ... 805
`frame-parameters` ... 805
`frame-parent` ... 836
`frame-pointer-visible-p` ... 839
`frame-position` ... 800
`frame-predicate`, a buffer display action alist entry ... 733
frame-relative coordinate ... 772
`frame-resize-pixelwise` ... 802
`frame-restack` ... 832
`frame-root-window` ... 695
`frame-scroll-bar-height` ... 1200
`frame-scroll-bar-width` ... 1200
`frame-selected-window` ... 699
`frame-size-changed-p` ... 803
`frame-terminal` ... 787
`frame-text-height` ... 802
`frame-text-width` ... 802
`frame-title-format` ... 824
`frame-visible-p` ... 831
`frame-width` ... 801
`frame-window-state-change` ... 785
`framep` ... 787
frames, scanning all ... 825
frameset ... 943
free list ... 1355
free variable ... 206
free variable, byte-compiler warning ... 322
`free_global_ref` ... 1377
frequency counts ... 357
`frexp` ... 40
fringe bitmaps ... 1197
fringe bitmaps, customizing ... 1198
fringe cursors ... 1196
fringe indicators ... 1195
`fringe-bitmaps-at-pos` ... 1198
`fringe-cursor-alist` ... 1196
`fringe-indicator-alist` ... 1195
fringes ... 1193
fringes, and empty line indication ... 1195
`fringes-outside-margins` ... 1194
front-sticky text property ... 935
`fround` ... 48
`fset` ... 249
`ftp-login` ... 188
`ftruncate` ... 48
full keymap ... 482
full-height window ... 702
full-width window ... 702
fullboth frames ... 811
fullheight frames ... 811
`fullscreen`, a frame parameter ... 811
`fullscreen-restore`, a frame parameter ... 811
fullwidth frames ... 811
`func-arity` ... 231
`funcall` ... 239, 1379
`funcall`, and debugging ... 344
`funcall-interactively` ... 434
`function` ... 244
function aliases ... 238
function call ... 148
function call debugging ... 337
function cell ... 133

function cell in autoload 306
function declaration 265
function definition 236
function descriptions 4
function form evaluation 148
function groups 607
function input stream 373
function invocation 239
function keys 442
function name 236
function not known to be defined, compilation warning 322
function output stream 376
function quoting 244
function safety 266
function's documentation string 235
`function-alias-p` 239
`function-documentation` 598
`function-documentation` property 597
`function-get` 139
`function-history` (function symbol property) 313
function-key-map 506
`function-put` 139
functionals 241
`functionp` 231
functions in modes 530
functions, making them interactive 425
`fundamental-mode` 528
`fundamental-mode-abbrev-table` 1074
future history in minibuffer input 387

G

gamma correction 821
`gap-position` 691
`gap-size` 691
garbage collection 1355
garbage collection protection 1363
`garbage-collect` 1356
`garbage-collection-messages` 1358
`gc-cons-percentage` 1358
`gc-cons-percentage`, in batch mode 1295
`gc-cons-threshold` 1358
`gc-elapsed` 1359
`gcs-done` 1359
generalized variable 224
`generate-new-buffer` 687
`generate-new-buffer-name` 677
`generated-autoload-file` 308
generators 175
generic commands 432
generic functions 245
generic mode 542
`gensym` 136
geometry specification 822
`get` 138
get node, tree-sitter 1048
`get`, `defcustom` keyword 283
`get-buffer` 676
`get-buffer-create` 687
`get-buffer-process` 1103
`get-buffer-window` 723
`get-buffer-window-list` 723
`get-buffer-xwidgets` 1234
`get-byte` 971
`get-char-code-property` 975
`get-char-property` 920
`get-char-property-and-overlay` 920
`get-charset-property` 977
`get-device-terminal` 789
`get-display-property` 1203
`get-file-buffer` 678
`get-internal-run-time` 1285
`get-largest-window` 721
`get-load-suffixes` 302
`get-lru-window` 721
`get-mru-window` 721
`get-pos-property` 920
`get-process` 1096
`get-register` 943
`get-text-property` 920
`get-unused-category` 1040
`get-variable-watchers` 200
`get-window-with-predicate` 721
`get_function_finalizer` 1371
`get_user_finalizer` 1378
`getenv` 1271
`gethash` 129
GID 1275
global binding 190
global break condition 352
global keymap 491
global variable 189
`global-abbrev-table` 1074
`global-buffers-menu-map` 1401
`global-disable-point-adjustment` 441
`global-key-binding` 504
`global-map` 492
`global-minor-modes` 545
`global-mode-string` 557
`global-set-key` 502
`global-unset-key` 503
glyph 1250
glyph code 1250
`glyph-char` 1250
`glyph-face` 1250
glyphless characters 1250
`glyphless-char` face 1251
`glyphless-char-display` 1251
`glyphless-char-display-control` 1251
`glyphless-display-mode` 1250
GNU ELPA 1312
gnutls cryptographic functions 950
gnutls cryptography inputs format 950
`gnutls-ciphers` 951

gnutls-digests ... 950
gnutls-hash-digest ... 950
gnutls-hash-mac ... 951
gnutls-macs ... 950
gnutls-symmetric-decrypt ... 951
gnutls-symmetric-encrypt ... 951
goto-char ... 856
goto-history-element ... 419
goto-line-history ... 395
goto-map ... 489
grammar, SMIE ... 585
grand-parent ... 594
grapheme cluster ... 1165
graphical display ... 787
graphical terminal ... 787
grave accent, quoting in documentation strings ... 1344
great-grand-parent ... 594
group, customization keyword ... 278
group-function, in completion ... 411
group-gid ... 1275
group-name ... 1275
group-real-gid ... 1275
groups of functions ... 607
gui-get-selection ... 843
gui-set-selection ... 842
guidelines for buffer display ... 745
gv-define-expander ... 226
gv-define-setter ... 226
gv-define-simple-setter ... 226
gv-letplace ... 227

H

hack-connection-local-variables ... 220
hack-connection-local-variables-apply ... 220
hack-dir-local-variables ... 217
hack-dir-local-variables-non-file-buffer ... 217
hack-local-variables ... 214
hack-local-variables-hook ... 215
Hamming weight ... 51
handle Lisp errors ... 182
handle-focus-in ... 829
handle-shift-selection ... 878
handle-switch-frame ... 829
handling errors ... 182
hardening ... 1305
has error, tree-sitter node ... 1055
hash code ... 130
hash notation ... 8
hash table access ... 129
hash tables ... 127
hash, cryptographic ... 947, 949
hash-table-count ... 132
hash-table-p ... 131
hash-table-rehash-size ... 132
hash-table-rehash-threshold ... 132
hash-table-size ... 132
hash-table-test ... 132
hash-table-weakness ... 132
hashing ... 135
header comments ... 1348
header line (of a window) ... 560
header line alignment when line numbers are displayed ... 1207
header-line, prefix key ... 463
header-line-format ... 560
header-line-format, a window parameter ... 781
header-line-format, and :align-to ... 1207
header-line-indent ... 560
header-line-indent-mode ... 560
header-line-indent-width ... 561
height of a line ... 1166
height of a window ... 701
height spec ... 1167
height, a frame parameter ... 810
help for major mode ... 535
help functions ... 604
help-buffer ... 606
help-char ... 605
help-command ... 605
help-echo (button property) ... 1237
help-echo (overlay property) ... 1158
help-echo (text property) ... 928
help-echo event ... 454
help-echo text, avoid command-key substitution ... 929
help-echo, customization keyword ... 293
help-echo-inhibit-substitution (text property) ... 929
help-event-list ... 605
help-form ... 605
help-key-binding (face) ... 601
help-map ... 605
help-setup-xref ... 606
help-window-select ... 606
Helper-describe-bindings ... 606
Helper-help ... 606
Helper-help-map ... 606
hex numbers ... 38
hidden buffers ... 676
history list ... 393
history of commands ... 478
history-add-new-input ... 394
history-delete-duplicates ... 394
history-length ... 394
HOME environment variable ... 1081
hook variables, list of ... 1403
hooks ... 525
hooks for changing a character ... 932
hooks for loading ... 314
hooks for motion of point ... 933
hooks for text changes ... 964
hooks for window operations ... 782
horizontal combination ... 695

horizontal position ... 912
horizontal scrolling ... 770
`horizontal-lineto` ... 1222
`horizontal-scroll-bar` ... 1201
`horizontal-scroll-bar`, prefix key ... 463
`horizontal-scroll-bar-mode` ... 1202
`horizontal-scroll-bars`, a frame parameter ... 812
`horizontal-scroll-bars-available-p` ... 1199
host language, tree-sitter ... 1062
how to visit files ... 610
HTML DOM ... 956
hyper characters ... 14
hyperlinks in documentation strings ... 1344

I

icon keywords ... 1232
icon types ... 1232
`icon-elements` ... 1232
`icon-left`, a frame parameter ... 809
`icon-name`, a frame parameter ... 817
`icon-string` ... 1232
`icon-title-format` ... 824
`icon-top`, a frame parameter ... 809
`icon-type`, a frame parameter ... 817
iconified frame ... 831
`iconify-child-frame` ... 837
`iconify-frame` ... 831
`iconify-frame` event ... 451
identical-contents objects, and byte-compiler ... 35
`identity` ... 241
idle timers ... 1289
idleness ... 1289
IEEE floating point ... 39
`if` ... 159
`if-let` ... 161
`ignore` ... 241
`ignore-error` ... 185
`ignore-errors` ... 185
`ignore-window-parameters` ... 780
`ignored-local-variable-values` ... 215
`ignored-local-variables` ... 216
image animation ... 1229
image cache ... 1230
image descriptor ... 1212
image formats ... 1211
image frames ... 1229
image maps ... 1215
image slice ... 1227
image types ... 1211
`image-animate` ... 1229
`image-animate-timer` ... 1230
`image-at-point-p` ... 1228
`image-cache-eviction-delay` ... 1230
`image-cache-size` ... 1231
`image-compute-scaling-factor` ... 1215
`image-crop` ... 1229
`image-current-frame` ... 1229
`image-cut` ... 1229
`image-decrease-size` ... 1228
`image-default-frame-delay` ... 1229
`image-flip-horizontally` ... 1229
`image-flip-vertically` ... 1229
`image-flush` ... 1230
`image-format-suffixes` ... 1217
`image-increase-size` ... 1228
`image-load-path` ... 1226
`image-load-path-for-library` ... 1226
`image-mask-p` ... 1216
`image-minimum-frame-delay` ... 1229
`image-multi-frame-p` ... 1229
`image-property` ... 1226
`image-rotate` ... 1229
`image-save` ... 1229
`image-scaling-factor` ... 1227
`image-show-frame` ... 1229
`image-size` ... 1228
`image-transforms-p` ... 1216
`image-type-available-p` ... 1211
`image-types` ... 1211
ImageMagick images ... 1217
`imagemagick-enabled-types` ... 1217
`imagemagick-types` ... 1217
`imagemagick-types-inhibit` ... 1217
images in buffers ... 1211
images, support for more formats ... 1217
Imenu ... 562
`imenu-add-to-menubar` ... 562
`imenu-case-fold-search` ... 563
`imenu-create-index-function` ... 563
`imenu-extract-index-name-function` ... 563
`imenu-generic-expression` ... 562
`imenu-prev-index-position-function` ... 563
`imenu-syntax-alist` ... 563
implicit `progn` ... 158
implied frame resizing ... 803
implied resizing of frame ... 803
inactive minibuffer ... 387
`inc` ... 269
indefinite extent ... 201
`indent-according-to-mode` ... 914
`indent-code-rigidly` ... 916
`indent-for-tab-command` ... 913
`indent-line-function` ... 914
`indent-region` ... 915
`indent-region-function` ... 916
`indent-relative` ... 916
`indent-relative-first-indent-point` ... 917
`indent-rigidly` ... 916
`indent-tabs-mode` ... 913
`indent-to` ... 913
`indent-to-left-margin` ... 906
indentation ... 913
indentation rules, for parser-based indentation ... 592

indentation rules, SMIE 588
`indicate-buffer-boundaries` 1195
`indicate-empty-lines` 1195
indicators, fringe 1195
indirect buffers 689
indirect specifications 362
`indirect-function` 148
`indirect-variable` 223
indirection for functions 147
infinite loops 337
infinity 40
information of node, syntax trees 1054
inheritance, for faces 1171
inheritance, keymap 488
inheritance, syntax table 1025
inheritance, text property 935
inhibit asynchronous native compilation 332
`inhibit-default-init` 1264
`inhibit-double-buffering`, a frame parameter 817
`inhibit-eol-conversion` 990
`inhibit-field-text-motion` 857
`inhibit-file-name-handlers` 654
`inhibit-file-name-operation` 654
`inhibit-interaction` 422
`inhibit-isearch` (text property) 930
`inhibit-iso-escape-detection` 984
`inhibit-local-variables-regexps` 214, 533
`inhibit-message` 1137
`inhibit-message-regexps` 1137
`inhibit-modification-hooks` 965
`inhibit-null-byte-detection` 984
`inhibit-point-motion-hooks` 934
`inhibit-quit` 473
`inhibit-read-only` 682
`inhibit-read-only` (text property) 930
`inhibit-same-window`, a buffer display action alist entry 733
`inhibit-splash-screen` 1262
`inhibit-startup-echo-area-message` 1262
`inhibit-startup-message` 1262
`inhibit-startup-screen` 1262
`inhibit-switch-frame`, a buffer display action alist entry 734
`inhibit-x-resources` 852
init file 1263
`init-file-user` 1274
`init.el` 1263
`initial-buffer-choice` 1262
`initial-environment` 1272
`initial-frame-alist` 806
`initial-major-mode` 534
`initial-scratch-message` 1263
`initial-window-system` 1253
`initial-window-system`, and startup 1260
initialization of Emacs 1260
`initialize`, `defcustom` keyword 283
initiating drag-and-drop 846
initiating drag-and-drop, low-level 847
inline completion 412
inline functions 261
`inline-const-p` 263
`inline-const-val` 263
`inline-error` 263
`inline-letevals` 262
`inline-quote` 262
inner edges 797
inner frame 797
inner height 797
inner size 797
inner width 797
innermost containing parentheses 1036
input devices 439
input events 441
input events, prevent recording 469
input focus 827
input methods 994
input modes 1290
input stream 372
`input-decode-map` 505
`input-method-alist` 994
`input-method-function` 468
`input-pending-p` 470
insecure text 948
`insert` 885
`insert-abbrev-table-description` 1070
`insert-and-inherit` 936
`insert-before-markers` 885
`insert-before-markers-and-inherit` 936
`insert-behind-hooks` (overlay property) 1159
`insert-behind-hooks` (text property) 933
`insert-buffer` 886
`insert-buffer-substring` 885
`insert-buffer-substring-as-yank` 893
`insert-buffer-substring-no-properties` 886
`insert-button` 1238
`insert-char` 885
`insert-default-directory` 407
`insert-directory` 650
`insert-directory-program` 650
`insert-file-contents` 616
`insert-file-contents-literally` 617
`insert-for-yank` 893
`insert-image` 1227
`insert-in-front-hooks` (overlay property) 1159
`insert-in-front-hooks` (text property) 933
`insert-into-buffer` 886
`insert-register` 943
`insert-sliced-image` 1227
`insert-text-button` 1239
inserting killed text 894
insertion before point 884
insertion of text 884
insertion type of a marker 874
inside comment 1036
inside string 1036

inspection of tree-sitter parse tree nodes 1044
`installation-directory` 1273
instrumenting for Edebug 347
`int-to-string` 65
`intangible` (overlay property) 1159
`intangible` (text property) 930
integer range 39
integer to decimal 65
integer to hexadecimal 67
integer to octal 67
integer to string 65
integer types (C programming language) 1393
`integer-or-marker-p` 872
`integer-width` 39
`integerp` 41
integers 38
integers in specific radix 38
interaction parameters between frames 814
`interactive` 425
interactive call 433
interactive code description 428
interactive completion 428
interactive function 425
interactive spec, using 425
interactive specification in primitives 1362
`interactive`, examples of using 431
`interactive-form` 427
`interactive-form` property 425
`interactive-form`, symbol property 425
`interactive-only` property 425
`intern` 137, 1379
`intern-soft` 137
internal menu bar 795
internal representation of characters 966
internal tool bar 795
internal windows 692
`internal-border-width`, a frame parameter ... 812
internals, of buffer 1383
internals, of process 1392
internals, of window 1387
interning 135
interpreter 145
`interpreter-mode-alist` 534
`interprogram-cut-function` 896
`interprogram-paste-function` 896
interrupt Lisp functions 472
`interrupt-process` 1100
`interrupt-process-functions` 1102
intersection of sequences 113
intervals 941
`intervals-consed` 1360
invalid prefix key error 499
`invalid-function` 147
`invalid-read-syntax` 8
`invalid-regexp` 1007
`invert-face` 1177
`invisible` (overlay property) 1159
`invisible` (text property) 930
invisible frame 831
invisible text 1146
`invisible-p` 1148
invisible/intangible text, and point 440
`invocation-directory` 1273
`invocation-name` 1273
invoking input method 468
invoking lisp debugger 341
is this call interactive 436
`is_not_nil` 1379
`isnan` 40
`iso8601-parse` 1281
`ISO 8601 date/time strings` 1281
ISO week, in time formatting 1282
italic text 1169
`iter-close` 176
`iter-defun` 175
`iter-do` 176
`iter-lambda` 175
`iter-next` 176
`iter-yield` 176
`iter-yield-from` 176
iteration 174
iteration over vector or string 114

J

JavaScript Object Notation 958
`javascript-callback` xwidget events 451
jit-lock functions, debugging 573
`jit-lock-debug-mode` 573
`jit-lock-register` 573
`jit-lock-unregister` 573
joining lists 91
`json-available-p` 958
`json-insert` 959
`json-parse-buffer` 959
`json-parse-string` 959
`json-serialize` 958
JSON remote procedure call protocol 959
`jsonrpc-async-request` 960
`jsonrpc-connection` 960
`jsonrpc-connection-ready-p` 962
`jsonrpc-connection-receive` 960
`jsonrpc-connection-send` 960
`jsonrpc-error` 960
`jsonrpc-lambda` 961
`jsonrpc-notify` 960
`jsonrpc-process-connection` 961
`jsonrpc-request` 960
`jsonrpc-running-p` 960
`jsonrpc-shutdown` 960
JSON 958
JSONRPC 959
JSONRPC application interfaces 960
JSONRPC connection initargs 961
JSONRPC deferred requests 962
JSONRPC object format 961

JSONRPC process-based connections 961
jumbled display of bidirectional text 1257
`just-one-space` 891
`justify-current-line` 903

K

`kbd` 481
`kbd-macro-termination-hook` 480
`keep-ratio`, a frame parameter 815
`kept-new-versions` 664
`kept-old-versions` 664
key 481
key binding 482
key binding, conventions for 1339
key lookup 495
key sequence 481
key sequence error 499
key sequence input 462
key substitution sequence 600
key translation function 506
`key-binding` 503
`key-description` 603
`key-translate` 467
`key-translation-map` 506
`key-valid-p` 481
keyboard events 441
keyboard events in strings 460
keyboard events, data in 457
keyboard input 461
keyboard input decoding on X 995
keyboard macro execution 435
keyboard macro termination 1252
keyboard macro, terminating 470
keyboard macros 479
keyboard macros (Edebug) 349
`keyboard-coding-system` 993
`keyboard-quit` 474
`keyboard-translate` 503
`keyboard-translate-table` 467
keymap 481
`keymap` (button property) 1237
`keymap` (overlay property) 1160
`keymap` (text property) 929
keymap entry 495
keymap format 482
keymap in keymap 495
keymap inheritance 488
keymap inheritance from multiple maps 488
keymap of character 929
keymap of character (and overlays) 1160
keymap prompt string 483
`keymap-global-lookup` 497
`keymap-global-set` 508
`keymap-global-unset` 508
`keymap-local-lookup` 497
`keymap-local-set` 508
`keymap-local-unset` 508
`keymap-lookup` 496
`keymap-parent` 488
`keymap-prompt` 511
`keymap-set` 499
`keymap-set-after` 522
`keymap-unset` 499
`keymapp` 484
keymaps for translating events 505
keymaps in modes 530
keymaps, scanning 508
keymaps, standard 1400
keys in documentation strings 600
keys, reserved 1339
keystroke 481
keyword symbol 189
`keywordp` 190
kill command repetition 437
kill ring 891
`kill-all-local-variables` 211
`kill-append` 896
`kill-buffer` 688
`kill-buffer-hook` 688
`kill-buffer-hook` in temporary buffers 675
`kill-buffer-query-functions` 688
`kill-buffer-query-functions` in temporary buffers 675
`kill-emacs` 1267
`kill-emacs-hook` 1268
`kill-emacs-query-functions` 1268
`kill-local-variable` 210
`kill-new` 896
`kill-process` 1101
`kill-read-only-ok` 893
`kill-region` 892
`kill-ring` 897
`kill-ring-max` 898
`kill-ring-yank-pointer` 897
`kill-xwidget` 1233
killing buffers 687
killing Emacs 1267
kinship, syntax tree nodes 1050
`kmacro-keymap` 1401

L

labeled narrowing 869
labeled restriction 869
`lambda` 243
lambda expression 232
`lambda` in debug 342
`lambda` in keymap 496
lambda list 233
lambda-list (Edebug) 363
language argument, for tree-sitter 1042
language grammar version, compatibility 1043
language grammar, for tree-sitter 1042
`language-change` event 454
largest fixnum 39

largest window ... 721
last ... 81
last visible position in a window ... 762
last-abbrev ... 1073
last-abbrev-location ... 1073
last-abbrev-text ... 1073
last-buffer ... 684
last-coding-system-used ... 982
last-command ... 437
last-command-event ... 439
last-event-device ... 439
last-event-frame ... 439
last-input-event ... 470
last-kbd-macro ... 480
last-nonmenu-event ... 439
last-prefix-arg ... 475
last-repeatable-command ... 437
lax-plist-get ... 101
lax-plist-put ... 101
layout of frame ... 793
layout on display, and bidirectional text ... 1257
layout parameters of frames ... 812
lazy evaluation ... 155
lazy loading ... 320
lazy-completion-table ... 399
ldexp ... 41
leaf node, of tree-sitter parse tree ... 1049
leap seconds ... 1275
least recently used window ... 721
left and right window decorations ... 693
left position ratio ... 808
left, a frame parameter ... 808
left-fringe, a frame parameter ... 813
left-fringe, prefix key ... 463
left-fringe-width ... 1194
left-margin ... 906
left-margin, prefix key ... 463
left-margin-width ... 1210
length ... 102
length< ... 103
length= ... 103
length> ... 103
let ... 191
let* ... 191
let-alist ... 99
letrec ... 192
Levenshtein distance ... 64
lexical binding ... 201
lexical binding (Edebug) ... 354
lexical comparison of strings ... 61
lexical environment ... 203
lexical scope ... 201
lexical-binding ... 205
library ... 299
library compilation ... 319
library header comments ... 1348
library search ... 302
libxml-available-p ... 955
libxml-parse-html-region ... 955
libxml-parse-xml-region ... 955
line end conversion ... 980
line height ... 799, 1166
line number ... 860
line truncation ... 1134
line wrapping ... 1134
line-beginning-position ... 859
line-end-position ... 859
line-height (text property) ... 932, 1166
line-move-ignore-invisible ... 1148
line-number-at-pos ... 860
line-number-display-width ... 1166
line-pixel-height ... 1165
line-prefix ... 1135
line-spacing ... 1167
line-spacing (text property) ... 931, 1167
line-spacing, a frame parameter ... 813
lines ... 858
lines in region ... 860
lineto ... 1222
link, customization keyword ... 278
linked list ... 16
linking files ... 631
Lisp debugger ... 334
Lisp expression motion ... 863
Lisp history ... 1
Lisp library ... 299
Lisp nesting error ... 154
Lisp object ... 8
Lisp objects, stack-allocated ... 1359
Lisp package ... 1308
Lisp printer ... 379
Lisp reader ... 372
Lisp timestamp ... 1275
lisp variables defined in C, restrictions ... 223
lisp-directory ... 304
lisp-indent-function property ... 276
lisp-mode-abbrev-table ... 1074
lisp-mode-autoload-regexp ... 308
lisp-mode.el ... 543
list ... 82
list all coding systems ... 982
list elements ... 79
list form evaluation ... 147
list in keymap ... 495
list length ... 102
list modification ... 85
list motion ... 863
list of threads ... 1079
list predicates ... 78
list reverse ... 104
list structure ... 16, 77
list to vector ... 114
list, replace element ... 88
list-buffers-directory ... 679
list-charset-chars ... 977
list-fonts ... 1190

`list-load-path-shadows` ... 304
`list-multisession-values` ... 229
`list-processes` ... 1095
`list-system-processes` ... 1111
`list-threads` ... 1079
`list-timers` ... 1289
`listify-key-sequence` ... 470
listing all buffers ... 683
`listp` ... 78
lists ... 77
lists and cons cells ... 77
lists as sets ... 91
literal evaluation ... 146
literate programming ... 915
little endian, in bindat specification ... 1128
live buffer ... 688
live node, tree-sitter ... 1055
live windows ... 692
`ln` ... 633
`load` ... 299
load error with require ... 310
load errors ... 300
`load`, customization keyword ... 279
`load-average` ... 1273
`load-changed` xwidget event ... 450
'`load-committed`' in xwidgets ... 451
`load-file` ... 301
`load-file-name` ... 301
`load-file-rep-suffixes` ... 302
'`load-finished`' in xwidgets ... 450
`load-history` ... 313
`load-in-progress` ... 301
`load-library` ... 301
`load-path` ... 302
`load-prefer-newer` ... 302
`load-read-function` ... 301
'`load-redirected`' in xwidgets ... 450
'`load-started`' in xwidgets ... 450
`load-suffixes` ... 301
`load-theme` ... 297
`loaddefs-generate` ... 306
loading ... 299
loading and configuring features ... 312
loading hooks ... 314
loading language grammar for tree-sitter ... 1042
loading, and non-ASCII characters ... 305
`loadup.el` ... 1351
local binding ... 190
local keymap ... 490
local part of remote file name ... 655
local variables ... 190
local variables, killed by major mode ... 211
`local`, `defcustom` keyword ... 284
`local-abbrev-table` ... 1074
`local-function-key-map` ... 506
`local-key-binding` ... 504
`local-map` (overlay property) ... 1160
`local-map` (text property) ... 929
`local-minor-modes` ... 545
`local-set-key` ... 503
`local-unset-key` ... 503
`local-variable-if-set-p` ... 210
`local-variable-p` ... 210
locale ... 995
`locale-coding-system` ... 995
locale-dependent string comparison ... 62
locale-dependent string equivalence ... 61
`locale-info` ... 995
locate file in path ... 630
`locate-dominating-file` ... 649
`locate-file` ... 630
`locate-library` ... 304
`locate-user-emacs-file` ... 647
lock file ... 619
`lock-buffer` ... 620
`lock-file-mode` ... 621
`lock-file-name-transforms` ... 620
`log` ... 51
`logand` ... 49
`logb` ... 41
`logcount` ... 51
logging echo-area messages ... 1141
logical arithmetic ... 48
logical lines, moving by ... 858
logical order ... 1255
logical shift ... 49
`logior` ... 50
`lognot` ... 51
`logxor` ... 50
looking up abbrevs ... 1072
looking up fonts ... 1185
`looking-at` ... 1013
`looking-at-p` ... 1014
`looking-back` ... 1013
lookup tables ... 127
`lookup-key` ... 503
loops, infinite ... 337
low-level key bindings ... 502
lower case ... 73
`lower-frame` ... 832
lowering a frame ... 832
LRO ... 1258
`lru-frames`, a buffer display action alist entry ... 736
`lru-time`, a buffer display action alist entry ... 736
`lsh` ... 49
`lwarn` ... 1143

M

M-g ... 489
M-s ... 489
M-S-x ... 435
M-x ... 435
M-X ... 435
Maclisp ... 1
macro ... 230
macro argument evaluation ... 273
macro call ... 269
macro call evaluation ... 149
macro caveats ... 272
macro compilation ... 318
macro descriptions ... 4
macro expansion ... 270
macro, how to define ... 271
`macroexpand` ... 270
`macroexpand-1` ... 270
`macroexpand-all` ... 270
`macrop` ... 269
macros ... 269
macros, at compile time ... 322
magic autoload comment ... 306
magic file names ... 651
`magic-fallback-mode-alist` ... 534
`magic-mode-alist` ... 534
`mail-host-address` ... 1271
main window ... 754
main window of a frame ... 754
`main-thread` ... 1077
major mode ... 527
major mode command ... 527
major mode conventions ... 529
major mode hook ... 531
major mode keymap ... 490
major mode, automatic selection ... 532
major mode, developing with tree-sitter ... 1064
`major-mode` ... 529
`major-mode-restore` ... 528
`major-mode-suspend` ... 528
`make-abbrev-table` ... 1069
`make-auto-save-file-name` ... 667
`make-backup-file-name` ... 665
`make-backup-file-name-function` ... 662
`make-backup-files` ... 662
`make-bool-vector` ... 122
`make-button` ... 1238
`make-byte-code` ... 324
`make-category-set` ... 1040
`make-category-table` ... 1040
`make-char-table` ... 120
`make-composed-keymap` ... 488
`make-condition-variable` ... 1079
`make-directory` ... 650
`make-display-table` ... 1248
`make-empty-file` ... 651
`make-finalizer` ... 25
`make-frame` ... 788
`make-frame-invisible` ... 831
`make-frame-on-display` ... 791
`make-frame-on-monitor` ... 792
`make-frame-visible` ... 831
`make-frame-visible` event ... 452
`make-glyph-code` ... 1250
`make-hash-table` ... 127
`make-help-screen` ... 606
`make-indirect-buffer` ... 689
`make-keymap` ... 485
`make-list` ... 82
`make-local-variable` ... 208
`make-marker` ... 873
`make-multisession` ... 229
`make-mutex` ... 1078
`make-nearby-temp-file` ... 645
`make-network-process` ... 1118
`make-obsolete` ... 260
`make-obsolete-generalized-variable` ... 227
`make-obsolete-variable` ... 222
`make-overlay` ... 1154
`make-pipe-process` ... 1092
`make-process` ... 1089
`make-progress-reporter` ... 1139
`make-record` ... 125
`make-ring` ... 123
`make-serial-process` ... 1126
`make-sparse-keymap` ... 484
`make-string` ... 55
`make-symbol` ... 136
`make-symbolic-link` ... 633
`make-syntax-table` ... 1030
`make-temp-file` ... 644
`make-temp-name` ... 645
`make-text-button` ... 1239
`make-thread` ... 1076
`make-translation-table` ... 978
`make-translation-table-from-alist` ... 979
`make-translation-table-from-vector` ... 979
`make-variable-buffer-local` ... 209
`make-vector` ... 118
`make-vtable` ... 539
`make-xwidget` ... 1233
`make_big_integer` ... 1374
`make_float` ... 1376
`make_function` ... 1369
`make_global_ref` ... 1377
`make_integer` ... 1374
`make_interactive` ... 1370
`make_string` ... 1377
`make_time` ... 1377
`make_unibyte_string` ... 1377
`make_user_ptr` ... 1378
making backup files ... 661
making buttons ... 1238
`makunbound` ... 193
malicious use of directional overrides ... 1258
managing overlays ... 1154

manipulating buttons ... 1239
map-char-table ... 121
map-charset-chars ... 977
map-keymap ... 509
map-y-or-n-p ... 416
mapatoms ... 137
mapbacktrace ... 345
mapc ... 243
mapcan ... 242
mapcar ... 242
mapconcat ... 243
maphash ... 130
mapped frame ... 831
mapping functions ... 241
margins, display ... 1209
margins, filling ... 905
mark ... 876
mark excursion ... 867
mark ring ... 876
mark, the ... 875
mark-active ... 878
mark-even-if-inactive ... 877
mark-marker ... 876
mark-ring ... 878
mark-ring-max ... 879
marker argument ... 430
marker creation ... 872
marker garbage collection ... 871
marker information ... 874
marker input stream ... 372
marker output stream ... 376
marker relocation ... 871
marker, how to move position ... 875
marker-buffer ... 874
marker-insertion-type ... 875
marker-position ... 874
markerp ... 872
markers ... 871
markers as numbers ... 871
markers, predicates for ... 872
match ... 593
match data ... 1014
match, customization keyword ... 293
match-alternatives, customization keyword ... 291
match-beginning ... 1017
match-buffers ... 686
match-data ... 1018
match-end ... 1017
match-inline, customization keyword ... 293
match-string ... 1016
match-string-no-properties ... 1017
match-substitute-replacement ... 1016
matching, structural ... 171
mathematical functions ... 51
max ... 43
max-char ... 971
max-image-size ... 1228
max-lisp-eval-depth ... 154
max-mini-window-height ... 420
maximize-window ... 708
maximized frames ... 811
maximizing windows ... 708
maximum fixnum ... 39
maximum value of character codepoint ... 971
maximum value of sequence ... 114
maximum-scroll-margin ... 767
maybe_quit, use in Lisp primitives ... 1363
md5 ... 948
MD5 checksum ... 947, 949
measuring resource usage ... 370
member ... 93
member-ignore-case ... 94
membership in a list ... 92
memory allocation ... 1355
memory usage ... 370, 1360
memory-full ... 1359
memory-info ... 1359
memory-limit ... 1359
memory-report ... 1359
memory-use-counts ... 1359
memq ... 92
memql ... 93
menu bar ... 517
menu bar keymaps ... 1401
menu definition example ... 516
menu item ... 511
menu keymaps ... 511
menu modification ... 522
menu prompt string ... 511
menu separators ... 514
menu-bar, prefix key ... 463
menu-bar-file-menu ... 1401
menu-bar-final-items ... 518
menu-bar-help-menu ... 1401
menu-bar-lines, a frame parameter ... 813
menu-bar-options-menu ... 1401
menu-bar-tools-menu ... 1401
menu-bar-update-hook ... 518
menu-item ... 512
menu-prompt-more-char ... 516
menus, popup ... 839
merge-face-attribute ... 1176
message ... 1135
message digest ... 947
message, finding what causes a particular message ... 336
message-box ... 1138
message-log-max ... 1141
message-or-box ... 1138
message-truncate-lines ... 1142
messages-buffer ... 1141
messages-buffer-name ... 1141
meta character key constants ... 502
meta character printing ... 603
meta characters ... 14

meta characters lookup 483
meta-prefix-char 498
min 43
min-height, a frame parameter 811
min-margins, a window parameter 782
min-width, a frame parameter 811
minibuffer 386
minibuffer completion 399
minibuffer contents, accessing 421
minibuffer history 393
minibuffer input 476
minibuffer input, and
command-line arguments 1083
minibuffer input, reading lisp objects 392
minibuffer input, reading text strings 387
minibuffer window 694
minibuffer window, and next-window 719
minibuffer windows 419
minibuffer, a frame parameter 814
minibuffer-allow-text-properties 390
minibuffer-auto-raise 833
minibuffer-beginning-of-
buffer-movement 403
minibuffer-complete 402
minibuffer-complete-and-exit 402
minibuffer-complete-word 402
minibuffer-completion-confirm 401
minibuffer-completion-help 402
minibuffer-completion-predicate 401
minibuffer-completion-table 401
minibuffer-confirm-exit-commands 402
minibuffer-contents 421
minibuffer-contents-no-properties 421
minibuffer-default-add-function 394
minibuffer-default-prompt-format 391
minibuffer-depth 421
minibuffer-exit, a frame parameter 815
minibuffer-exit-hook 423
minibuffer-frame-alist 806
minibuffer-help-form 423
minibuffer-history 395
minibuffer-inactive-mode 423
minibuffer-inactive-mode-map 1401
minibuffer-less frame 797
minibuffer-local-completion-map 403
minibuffer-local-filename-
completion-map 403
minibuffer-local-map 390
minibuffer-local-must-match-map 403
minibuffer-local-ns-map 391
minibuffer-local-shell-command-map 408
minibuffer-message 423
minibuffer-message (text property) 934
minibuffer-message-timeout 423
minibuffer-mode-map 390
minibuffer-only frame 797, 806
minibuffer-prompt 421
minibuffer-prompt-end 421
minibuffer-prompt-properties 388
minibuffer-prompt-width 421
minibuffer-scroll-window 423
minibuffer-selected-window 423
minibuffer-setup-hook 422
minibuffer-window 419
minibuffer-window-active-p 420
minibuffer-with-setup-hook 423
minibufferp 422
minimize-window 708
minimized frame 831
minimizing windows 708
minimum fixnum 39
minimum value of sequence 114
minor mode 544
minor mode conventions 545
minor-mode-alist 557
minor-mode-key-binding 498
minor-mode-list 545
minor-mode-map-alist 493
minor-mode-overriding-map-alist 493
mirroring of characters 973
missing node, tree-sitter 1054
mkdir 650
mod 47
mode 525
mode bits 623
mode help 535
mode hook 531
mode line 551
mode line construct 552
mode loading 532
mode variable 545
mode, a buffer display action alist entry 733
mode-class (property) 532
mode-line, prefix key 463
mode-line-buffer-identification 555
mode-line-client 556
mode-line-coding-system-map 1401
mode-line-column-line-number-mode-map ... 1401
mode-line-compact 552
mode-line-end-spaces 556
mode-line-format 554
mode-line-format, a window parameter 781
mode-line-frame-identification 555
mode-line-front-space 556
mode-line-input-method-map 1401
mode-line-misc-info 556
mode-line-modes 555
mode-line-modified 555
mode-line-mule-info 555
mode-line-percent-position 555
mode-line-position 555
mode-line-position-column-format 556
mode-line-position-column-line-format 556
mode-line-position-line-format 556
mode-line-process 556
mode-line-remote 556

`mode-line-window-selected-p` ... 552
`mode-name` ... 556
mode-specific commands ... 431
`mode-specific-map` ... 489
model/view/controller ... 1241
modification flag (of buffer) ... 679
modification of lists ... 90
modification time of buffer ... 680
modification time of file ... 627
`modification-hooks` (overlay property) ... 1159
`modification-hooks` (text property) ... 932
modifier bits (of input character) ... 441
modifiers of events ... 467
modify a list ... 85
`modify-all-frames-parameters` ... 805
`modify-category-entry` ... 1041
`modify-frame-parameters` ... 805
`modify-syntax-entry` ... 1030
modifying strings ... 59
module API ... 1366
module functions ... 1368
module initialization ... 1366
module runtime environment ... 1367
module values, conversion ... 1371
`module-file-suffix` ... 315
`module-load` ... 316
modulus ... 47
`momentary-string-display` ... 1153
monitor change functions ... 793
most recently selected windows ... 699
most recently used window ... 721
`most-negative-fixnum` ... 39
`most-positive-fixnum` ... 39
motion based on parsing ... 1034
motion by chars, words, lines, lists ... 856
motion event ... 448
mouse click event ... 443
mouse drag event ... 446
mouse dragging parameters ... 816
mouse events, data in ... 457
mouse events, in special parts of window or frame ... 463
mouse events, repeated ... 447
mouse motion events ... 448
mouse pointer shape ... 841
mouse position ... 838
mouse position list ... 444
mouse position list, accessing ... 458
mouse tracking ... 837
mouse wheel event ... 443
`mouse`, a face ... 822
mouse, availability ... 852
mouse-1 ... 937
`mouse-1-click-follows-link` ... 938
mouse-2 ... 1339
`mouse-absolute-pixel-position` ... 839
`mouse-action` (button property) ... 1237
`mouse-appearance-menu-map` ... 1401
`mouse-autoselect-window` ... 776
`mouse-color`, a frame parameter ... 822
`mouse-face` (button property) ... 1237
`mouse-face` (overlay property) ... 1158
`mouse-face` (text property) ... 928
`mouse-fine-grained-tracking` ... 449
`mouse-leave-buffer-hook` ... 1405
`mouse-movement-p` ... 457
`mouse-on-link-p` ... 939
`mouse-pixel-position` ... 839
`mouse-position` ... 838
`mouse-position-function` ... 838
`mouse-wheel-down-event` ... 452
`mouse-wheel-frame`, a frame parameter ... 815
`mouse-wheel-left-event` ... 452
`mouse-wheel-right-event` ... 452
`mouse-wheel-up-event` ... 452
move to beginning or end of buffer ... 858
`move-frame-functions` ... 800
`move-marker` ... 875
`move-overlay` ... 1155
`move-point-visually` ... 1257
`move-to-column` ... 912
`move-to-left-margin` ... 906
`move-to-window-group-line` ... 862
`move-to-window-group-line-function` ... 862
`move-to-window-line` ... 861
`movemail` ... 1082
`moveto` ... 1222
moving across syntax classes ... 1033
moving markers ... 875
MS-DOS and file modes ... 623
MS-Windows file-name syntax ... 636
`mule-keymap` ... 489
multi-file package ... 1310
multi-frame images ... 1229
`multi-message-max` ... 1137
`multi-message-timeout` ... 1137
multi-mode indentation ... 915
multi-monitor ... 791
`multi-query-replace-map` ... 1022
multi-tty ... 789
multibyte characters ... 966
multibyte text ... 966
`multibyte-char-to-unibyte` ... 969
`multibyte-string-p` ... 967
`multibyte-syntax-as-symbol` ... 1037
multiline font lock ... 577
multiple languages, parsing with tree-sitter ... 1060
multiple terminals ... 789
multiple windows ... 692
multiple X displays ... 789
multiple yes-or-no questions ... 416
multiple-dispatch methods ... 248
`multiple-frames` ... 824
multisession variable ... 227
`multisession-delete` ... 229
`multisession-directory` ... 229

multisession-edit-mode ... 229
multisession-storage ... 229
multisession-value ... 228
mutable lists ... 87
mutable objects ... 37
mutex-lock ... 1078
mutex-name ... 1078
mutex-unlock ... 1078
mutexp ... 1078
mwheel-coalesce-scroll-events ... 452

N

n-p-gp ... 593
name, a frame parameter ... 807
named function ... 236
named node, tree-sitter ... 1044
named-let ... 192
namespace etiquette ... 142
namespaces ... 142
namespacing ... 142
naming backup files ... 665
NaN ... 40
narrow-map ... 1401
narrow-to-page ... 868
narrow-to-region ... 867
narrowing ... 867
native code ... 328
native compilation ... 328
native compilation, prevent writing *.eln files ... 328
native edges ... 796
native frame ... 796
native height ... 796
native position ... 796
native size ... 796
native width ... 796
native-comp-async-jobs-number ... 331
native-comp-async-query-on-exit ... 332
native-comp-async-report-warnings-errors ... 332
native-comp-available-p ... 330
native-comp-debug ... 331
native-comp-eln-load-path ... 304
native-comp-enable-subr-trampolines ... 332
native-comp-jit-compilation ... 332
native-comp-limple-mode ... 329
native-comp-speed ... 331
native-comp-verbose ... 329, 331
native-compilation functions ... 329
native-compilation variables ... 331
native-compile ... 329
native-compile, a Lisp feature ... 328
native-compile-async ... 330
natnump ... 42
natural numbers ... 42
nbutlast ... 82
nconc ... 91
negative infinity ... 40
negative-argument ... 476
nest frame ... 834
network byte ordering, in Bindat specification ... 1128
network connection ... 1114
network connection, encrypted ... 1115
network servers ... 1117
network service name, and default coding system ... 987
network-coding-system-alist ... 988
network-interface-info ... 1124
network-interface-list ... 1123
network-lookup-address-info ... 1124
network-stream-use-client-certificates ... 1116
new file message ... 613
newline ... 12, 887
newline and Auto Fill mode ... 887
newline in print ... 380
newline in strings ... 20
newline-and-indent ... 914
next input ... 469
next-button ... 1241
next-char-property-change ... 925
next-complete-history-element ... 419
next-frame ... 826
next-history-element ... 419
next-matching-history-element ... 419
next-overlay-change ... 1161
next-property-change ... 924
next-screen-context-lines ... 768
next-single-char-property-change ... 925
next-single-property-change ... 924
next-window ... 719
nil ... 2
nil as a list ... 17
nil in keymap ... 495
nil input stream ... 373
nil output stream ... 376
nlistp ... 78
no-accept-focus, a frame parameter ... 818
no-byte-compile ... 317
no-catch ... 178
no-conversion coding system ... 981
no-delete-other-windows, a window parameter ... 780
no-focus-on-map, a frame parameter ... 818
no-indent ... 595
no-native-compile ... 328
no-node ... 593
no-other-frame, a frame parameter ... 815
no-other-window, a window parameter ... 781
no-redraw-on-reenter ... 1133
no-self-insert property ... 1070
no-special-glyphs, a frame parameter ... 813
node types, in a syntax tree ... 1044
node, ewoc ... 1241

`node-is` ... 593
nodes, by field name ... 1051
nodes, by kinship ... 1050
nodes, by position ... 1051
non-ASCII characters ... 966
non-ASCII characters in loaded files ... 305
non-ASCII text in key bindings ... 508
non-capturing group ... 1005
non-GNU ELPA ... 1312
non-greedy repetition characters in regexp ... 1000
`non_local_exit_check` ... 1381
`non_local_exit_clear` ... 1381
`non_local_exit_get` ... 1381
`non_local_exit_signal` ... 1382
`non_local_exit_throw` ... 1381
nondirectory part (of file name) ... 637
`noninteractive` ... 1294
nonlocal exits ... 177
nonlocal exits, cleaning up ... 187
nonlocal exits, in modules ... 1380
nonprinting characters, reading ... 468
`noreturn` ... 370
normal hook ... 525
`normal-auto-fill-function` ... 908
`normal-backup-enable-predicate` ... 662
`normal-mode` ... 533
`not` ... 162
not recording input events ... 469
`not-modified` ... 680
notation ... 3
notifications, on desktop ... 1296
`notifications-close-notification` ... 1299
`notifications-get-capabilities` ... 1299
`notifications-get-server-information` ... 1299
`notifications-notify` ... 1296
notifiers, tree-sitter ... 1048
`nreverse` ... 105
`ns-appearance`, a frame parameter ... 818
`ns-transparent-titlebar`, a frame parameter ... 819
`ntake` ... 81
`nth` ... 80
`nth-sibling` ... 594
`nthcdr` ... 80
`null` ... 78
null bytes, and decoding text ... 984
`null-device` ... 1274
`num-input-keys` ... 463
`num-nonmacro-input-events` ... 465
`num-processors` ... 1096
number comparison ... 42
number conversions ... 43
number of bignum bits, limit on ... 39
`number-or-marker-p` ... 872
`number-sequence` ... 85
`number-to-string` ... 65
numbered backups ... 664
`numberp` ... 41
numbers ... 38
numeric prefix argument ... 474
numeric prefix argument usage ... 430
numerical RGB color specification ... 848

O

obarray ... 135, 137
obarray in completion ... 397
object ... 8
object internals ... 1382
object to string ... 380
`object-intervals` ... 921
objects with identical contents, and byte-compiler ... 35
`obsolete` (`declare` spec) ... 264
obsolete functions ... 260
`oclosure-define` ... 251
`oclosure-interactive-form` ... 427
`oclosure-lambda` ... 252
`oclosure-type` ... 252
oclosures ... 250
octal character code ... 13
octal character input ... 468
octal escapes ... 1247
octal numbers ... 38
old advices, porting ... 258
`old-selected-frame` ... 786
`old-selected-window` ... 786
`one-window-p` ... 721
`only-global-abbrevs` ... 1071
opacity, frame ... 821
open closures ... 250
`open-dribble-file` ... 1292
`open-network-stream` ... 1115
`open-paren-in-column-0-is-defun-start` ... 864
`open-termscript` ... 1292
`open_channel` ... 1380
OpenType font ... 1190
operating system environment ... 1270
operating system signal ... 1268
`operations` (property) ... 654
operations on images ... 1228
optimize regexp ... 1009
option descriptions ... 6
optional arguments ... 234
options on command line ... 1266
`options`, `defcustom` keyword ... 282
`or` ... 162
ordering of windows, cyclic ... 719
origin of display ... 797
`other-buffer` ... 684
`other-window` ... 720
`other-window`, a window parameter ... 781
`other-window-scroll-buffer` ... 767
outdated node, tree-sitter ... 1054
outer border ... 794
outer edges ... 794

outer frame ... 794
outer height ... 794
outer position ... 794
outer size ... 794
outer width ... 794
`outer-window-id`, a frame parameter ... 817
output from processes ... 1102
output stream ... 376
output variables, overriding ... 384
output-controlling variables ... 381
overall prompt string ... 483
`overflow-newline-into-fringe` ... 1196
overlay properties ... 1157
overlay, empty ... 1154
`overlay-arrow-position` ... 1199
`overlay-arrow-string` ... 1199
`overlay-arrow-variable-list` ... 1199
`overlay-buffer` ... 1155
`overlay-end` ... 1155
`overlay-get` ... 1157
`overlay-properties` ... 1157
`overlay-put` ... 1157
`overlay-start` ... 1154
`overlayp` ... 1154
overlays ... 1154
overlays, managing ... 1154
overlays, searching for ... 1161
`overlays-at` ... 1161
`overlays-in` ... 1161
overlined text ... 1170
override existing functions ... 238
override redirect frames ... 818
override spec (for a face) ... 1175
`override-redirect`, a frame parameter ... 818
overrides, in output functions ... 384
overriding bidirectional properties ... 1258
`overriding-local-map` ... 493
`overriding-local-map-menu-flag` ... 494
`overriding-terminal-local-map` ... 494
`overwrite-mode` ... 887

P

package ... 1308
package archive ... 1312
package archive security ... 1312
package attributes ... 1308
package autoloads ... 1309
package dependencies ... 1308
package name ... 1308
package signing ... 1312
package version ... 1308
`package-activate-all` ... 1309
`package-archives` ... 1312
`package-initialize` ... 1309
`package-version`, customization keyword ... 280
packing ... 1128
padding ... 69
`page-delimiter` ... 1023
`paragraph-separate` ... 1023
paragraph-separate, and bidirectional display ... 1256
`paragraph-start` ... 1023
paragraph-start, and bidirectional display ... 1256
parameters for moving frames with the mouse ... 816
parameters for resizing frames with the mouse ... 816
parameters of initial frame ... 806
`parent` ... 594
parent directory of file ... 649
parent frames ... 833
parent of char-table ... 119
parent process ... 1081
parent window ... 695
`parent-bol` ... 594
`parent-frame`, a buffer display action alist entry ... 737
`parent-frame`, a frame parameter ... 815
`parent-is` ... 593
parenthesis ... 16
parenthesis depth ... 1037
parenthesis matching ... 1246
parenthesis mismatch, debugging ... 369
parity, in serial connections ... 1127
parse state for a position ... 1035
parse string, tree-sitter ... 1048
`parse-colon-path` ... 1273
`parse-partial-sexp` ... 1036
`parse-sexp-ignore-comments` ... 1037
`parse-sexp-lookup-properties` ... 1032
`parse-time-string` ... 1281
parser state ... 1035
parser-based font-lock ... 580
parser-based indentation ... 592
parsing buffer text ... 1025
parsing expressions ... 1033
parsing html ... 955
parsing multiple languages with tree-sitter ... 1060
parsing program source ... 1042
parsing xml ... 955
parsing, control parameters ... 1037
partial application of functions ... 240
partial-width windows ... 1134
passwords, reading ... 418
`path-separator` ... 1272
`PATH` environment variable ... 1081
pattern matching with tree-sitter nodes ... 1055
pattern matching, programming style ... 163
pattern syntax, tree-sitter query ... 1056
patterns, tree-sitter, in string form ... 1059
PBM ... 1224
pcase ... 163, 164
pcase pattern ... 164
pcase, defining new kinds of patterns ... 170
`pcase-defmacro` ... 170

`pcase-dolist` ... 174
`pcase-lambda` ... 174
`pcase-let` ... 173
`pcase-let*` ... 173
`pcase-setq` ... 174
`pdumper-stats` ... 1353
peculiar error ... 186
peeking at input ... 469
percent symbol in mode line ... 552
`perform-replace` ... 1020
performance analysis ... 370
performance analysis (Edebug) ... 357
permanent local variable ... 211
`permanently-enabled-local-variables` ... 214
permissions, file ... 623, 634
persistent window parameters ... 779
phishing using directional overrides ... 1258
physical lines, moving by ... 858
piece of advice ... 252
`pinch` event ... 453
pipe, when to use for subprocess communications ... 1089
pixel height of a window ... 702
pixel width of a window ... 702
`pixel-fill-region` ... 902
`pixel-fill-width` ... 902
pixel-resolution wheel events ... 452
pixelwise, resizing windows ... 706
place form ... 224
`play-sound` ... 1293
`play-sound-file` ... 1293
`play-sound-functions` ... 1294
plist ... 99
plist access ... 100
plist vs. alist ... 100
`plist-get` ... 100
`plist-member` ... 101
`plist-put` ... 100
`plistp` ... 99
`plugin_is_GPL_compatible` ... 315
point ... 855
point excursion ... 866, 867
point in window ... 761
point with narrowing ... 855
`point-entered` (text property) ... 933
`point-left` (text property) ... 933
`point-marker` ... 873
`point-max` ... 856
`point-max-marker` ... 873
`point-min` ... 855
`point-min-marker` ... 873
`pointer` (text property) ... 931
pointer shape ... 841
pointers ... 16
polymorphism ... 245
`pop` ... 79
`pop-mark` ... 877
`pop-to-buffer` ... 725
`pop-up-frame-alist` ... 740
`pop-up-frame-function` ... 739
`pop-up-frame-parameters`, a buffer display action alist entry ... 737
`pop-up-frames` ... 739
`pop-up-frames`, replacement for ... 740
`pop-up-windows` ... 738
`pop-up-windows`, replacement for ... 740
popcount ... 51
port number, and default coding system ... 987
`pos-bol` ... 859
`pos-eol` ... 859
`pos-visible-in-window-group-p` ... 765
`pos-visible-in-window-group-p-function` ... 765
`pos-visible-in-window-p` ... 764
position (in buffer) ... 855
position argument ... 429
position in window ... 761
position of frame ... 793, 800
position of mouse ... 838
`position-bytes` ... 967
`position-symbol` ... 144
positive infinity ... 40
`posix-looking-at` ... 1014
`posix-search-backward` ... 1014
`posix-search-forward` ... 1014
`posix-string-match` ... 1014
`posn-actual-col-row` ... 459
`posn-area` ... 458
`posn-at-point` ... 460
`posn-at-x-y` ... 460
`posn-col-row` ... 458
`posn-image` ... 459
`posn-object` ... 459
`posn-object-width-height` ... 459
`posn-object-x-y` ... 459
`posn-point` ... 458
`posn-string` ... 459
`posn-timestamp` ... 459
`posn-window` ... 458
`posn-x-y` ... 458
`posnp` ... 458
`post-command-hook` ... 424
`post-gc-hook` ... 1358
`post-self-insert-hook` ... 886
`pp` ... 381
`pre-command-hook` ... 424
`pre-redisplay-function` ... 1134
`pre-redisplay-functions` ... 1134
precedence of buffer display action functions ... 741
`preceding-char` ... 881
precision in format specifications ... 69
predicates for lists ... 78
predicates for markers ... 872
predicates for numbers ... 41
predicates for strings ... 55
predicates for syntax tree nodes ... 1054
`preedit-text` event ... 453

`prefer-utf-8` coding system 980
prefix argument 474
prefix argument unreading 469
prefix command 490
prefix key 489
`prefix`, `defgroup` keyword 281
`prefix-arg` 475
`prefix-command-echo-keystrokes-functions` 1405
`prefix-command-preserve-state-hook` 1406
`prefix-help-command` 605
`prefix-numeric-value` 475
preloaded Lisp files 1352
`preloaded-file-list` 1352
preloading additional functions and variables 1352
`prepare-change-group` 962
`preserve-size`, a buffer display action alist entry 736
preserving window sizes 709
pretty-printer 381
`prev-adaptive-prefix` 595
`prev-line` 595
`prev-sibling` 594
prevent warnings in init files 1145
preventing backtracking 362
preventing prefix key 496
preventing quitting 473
previous complete subexpression 1036
`previous-button` 1241
`previous-char-property-change` 925
`previous-complete-history-element` 419
`previous-frame` 826
`previous-history-element` 419
`previous-matching-history-element` 419
`previous-overlay-change` 1161
`previous-property-change` 924
`previous-single-char-property-change` 925
`previous-single-property-change` 925
`previous-window` 720
`previous-window`, a buffer display action alist entry 733
primary selection 842
primitive 230
primitive function 24
primitive function internals 1361
primitive type 8
`primitive-undo` 900
`prin1` 379
`prin1-to-string` 380
`princ` 379
`print` 379
print example 377
print name cell 133
`print-charset-text-property` 382
`print-circle` 383
`print-continuous-numbering` 383
`print-escape-control-characters` 381
`print-escape-multibyte` 382
`print-escape-newlines` 381
`print-escape-nonascii` 382
`print-gensym` 383
`print-integers-as-characters` 383
`print-length` 382
`print-level` 382
`print-number-table` 383
`print-quoted` 381
`print-symbols-bare` 144
`print-unreadable-function` 383
printable ASCII characters 1247
`printable-chars` 976
printed representation 8
printed representation for characters 11
printing 372
printing (Edebug) 355
printing circular structures 355
printing limits 382
printing notation 3
`priority` (overlay property) 1157
priority order of coding systems 990
process 1081
process creation 1081
process filter 1104
process filter multibyte flag 1107
process information 1095
process input 1099
process internals 1392
process output 1102
process sentinel 1109
process signals 1100
`process-adaptive-read-buffering` 1102
`process-attributes` 1111
`process-buffer` 1103
`process-coding-system` 1098
`process-coding-system-alist` 987
`process-command` 1096
`process-connection-type` 1094
`process-contact` 1096
`process-datagram-address` 1118
`process-environment` 1272
`process-error-pause-time` 1094
`process-exit-status` 1098
`process-file` 1086
`process-file-return-signal-string` 1087
`process-file-shell-command` 1088
`process-file-side-effects` 1087
`process-filter` 1106
`process-get` 1098
`process-id` 1097
`process-kill-buffer-query-function` 1103
`process-lines` 1089
`process-lines-ignore-status` 1089
`process-list` 1095
`process-live-p` 1098
`process-mark` 1103
`process-name` 1097

`process-plist` ... 1099
`process-put` ... 1098
`process-query-on-exit-flag` ... 1111
`process-running-child-p` ... 1100
`process-send-eof` ... 1099
`process-send-region` ... 1099
`process-send-string` ... 1099
`process-sentinel` ... 1110
`process-status` ... 1097
`process-thread` ... 1108
`process-tty-name` ... 1098
`process-type` ... 1098
`process_input` ... 1380
processes, threads ... 1108
processing of errors ... 181
processor run time ... 1285
`processp` ... 1081
profile ... 370
`profiler-find-profile` ... 371
`profiler-find-profile-other-window` ... 371
`profiler-report` ... 370
`profiler-report-compare-profile` ... 371
`profiler-report-describe-entry` ... 371
`profiler-report-find-entry` ... 371
`profiler-start` ... 370
`profiler-stop` ... 370
profiling ... 370
`prog-first-column` ... 915
`prog-indentation-context` ... 915
`prog-mode` ... 537
`prog-mode`, and `bidi-paragraph-direction` ... 1257
`prog-mode-hook` ... 537
prog-mode-map ... 1401
`prog1` ... 159
`prog2` ... 159
`progn` ... 158
program arguments ... 1082
program directories ... 1082
program name, and default coding system ... 987
programmed completion ... 410
programming conventions ... 1340
programming types ... 10
progress reporting ... 1138
`progress-reporter-done` ... 1140
`progress-reporter-force-update` ... 1140
`progress-reporter-update` ... 1139
prompt for file name ... 405
prompt string (of menu) ... 511
prompt string of keymap ... 483
proper list ... 77
`proper-list-p` ... 78
properties of text ... 919
`propertize` ... 923
property category of text character ... 927
property list ... 99
property list cell ... 133
property lists vs association lists ... 100
protect C variables from garbage collection ... 1363
protected forms ... 187
`provide` ... 311
`provide-theme` ... 296
providing features ... 310
pseudo window ... 1388
pty, when to use for subprocess communications ... 1089
pure function ... 230
`pure` property ... 141
pure storage ... 1354
`pure-bytes-used` ... 1354
`purecopy` ... 1354
`purify-flag` ... 1354
`push` ... 85
`push-button` ... 1240
`push-mark` ... 877
`put` ... 138
`put-char-code-property` ... 975
`put-charset-property` ... 977
`put-image` ... 1228
`put-text-property` ... 922
`puthash` ... 129

Q

`quadratic-bezier-curveto` ... 1223
quantify node, tree-sitter ... 1057
queries, compiling ... 1060
`query` ... 593
query functions, tree-sitter ... 1056
query, tree-sitter ... 1056
`query-font` ... 1192
`query-replace-history` ... 395
`query-replace-map` ... 1021
querying the user ... 414
question mark in character constant ... 11
`quietly-read-abbrev-file` ... 1071
`quit-flag` ... 473
`quit-process` ... 1101
`quit-restore`, a window parameter ... 781
`quit-restore-window` ... 752
`quit-window` ... 752
`quit-window-hook` ... 752
quitting ... 472
quitting from infinite loop ... 337
quitting windows ... 752
`quote` ... 151
quote character ... 1036
quote special characters in regexp ... 1008
quoted character input ... 468
`quoted-insert` suppression ... 501
quoting and unquoting command-line arguments ... 1083
quoting apostrophe and grave accent in doc strings ... 1344
quoting characters in printing ... 378
quoting using apostrophe ... 151

R

radio, customization types ... 289
radix for reading an integer ... 38
`raise-frame` ... 832
raising a frame ... 832
`random` ... 52
random numbers ... 52
`rassoc` ... 96
`rassq` ... 97
`rassq-delete-all` ... 98
raw prefix argument ... 474
raw prefix argument usage ... 430
raw syntax descriptor ... 1037
`raw-text` coding system ... 980
`re-builder` ... 999
`re-search-backward` ... 1012
`re-search-forward` ... 1011
`read` ... 375
read command name ... 435
read file names ... 405
read input ... 461
read syntax ... 8
read syntax for characters ... 11
`read-answer` ... 417
`read-answer-short` ... 417
`read-buffer` ... 403
`read-buffer-completion-ignore-case` ... 404
`read-buffer-function` ... 404
`read-char` ... 464
`read-char-choice` ... 466
`read-char-choice-use-read-key` ... 466
`read-char-exclusive` ... 465
`read-char-from-minibuffer` ... 418
`read-circle` ... 375
`read-coding-system` ... 986
`read-color` ... 405
`read-command` ... 404
`read-directory-name` ... 407
`read-event` ... 464
`read-expression-history` ... 395
`read-extended-command-predicate` ... 432, 435
`read-file-modes` ... 635
`read-file-name` ... 406
`read-file-name-completion-ignore-case` ... 407
`read-file-name-function` ... 407
`read-from-minibuffer` ... 387
`read-from-string` ... 375
`read-hide-char` ... 418
`read-input-method-name` ... 994
`read-kbd-macro` ... 604
`read-key` ... 465
`read-key-sequence` ... 462
`read-key-sequence-vector` ... 463
`read-minibuffer` ... 392
`read-minibuffer-restore-windows` ... 392
`read-multiple-choice` ... 466
`read-no-blanks-input` ... 391
`read-non-nil-coding-system` ... 986
`read-number-history` ... 395
`read-only` (text property) ... 929
read-only buffer ... 682
read-only buffers in interactive ... 426
read-only character ... 929
read-only variables ... 190
`read-only-mode` ... 682
`read-passwd` ... 418
`read-positioning-symbols` ... 375
`read-quoted-char` ... 468
`read-quoted-char` quitting ... 473
`read-regexp` ... 389
`read-regexp-case-fold-search` ... 390
`read-regexp-defaults-function` ... 390
`read-shell-command` ... 408
`read-string` ... 388
`read-string-from-buffer` ... 389
`read-symbol-shorthands` ... 142
`read-variable` ... 405
`read-variable`, history list ... 395
readable syntax ... 376
`readablep` ... 376
reading ... 372
reading a single event ... 463
reading from files ... 616
reading from minibuffer with completion ... 399
reading grammar definition, tree-sitter ... 1045
reading interactive arguments ... 428
reading numbers in hex, octal, and binary ... 38
reading order ... 1255
reading symbols ... 135
`real-last-command` ... 437
rear-nonsticky text property ... 935
rear-nonsticky, and
  cursor-intangible property ... 930
rearrangement of lists ... 90
rebinding ... 498
`recent-auto-save-p` ... 668
`recent-keys` ... 1291
`recenter` ... 768
`recenter-positions` ... 769
`recenter-redisplay` ... 769
`recenter-top-bottom` ... 769
`recenter-window-group` ... 769
`recenter-window-group-function` ... 769
recombining windows ... 714
`record` ... 125
record command history ... 434
recording input ... 1291
`recordp` ... 125
records ... 125
rectangle, as contents of a register ... 943
recursion ... 174
`recursion-depth` ... 477
recursive command loop ... 476
recursive editing level ... 476
recursive evaluation ... 145
recursive minibuffers ... 421

recursive traverse of directory tree ... 648
recursive-edit ... 477
redefine existing functions ... 238
redirect-frame-focus ... 829
redisplay ... 1133
redisplay errors ... 336
redo ... 898
redraw-display ... 1133
redraw-frame ... 1133
reducing sequences ... 110
reference to free variable, compilation warning ... 322
references, following ... 1339
refresh the screen ... 1133
regexp ... 999
regexp alternative ... 1005
regexp grouping ... 1005
regexp searching ... 1011
regexp stack overflow ... 1010
regexp syntax ... 999
regexp, special characters in ... 999
regexp-history ... 395
regexp-opt ... 1009
regexp-opt-charset ... 1010
regexp-opt-depth ... 1010
regexp-quote ... 1008
regexp-unmatchable ... 1010
regexps used standardly in editing ... 1023
region ... 879
region argument ... 430
region-beginning ... 879
region-end ... 879
register preview ... 944
register-alist ... 943
register-definition-prefixes ... 308
register-read-with-preview ... 944
registers ... 942
regular expression ... 999
regular expression problems ... 1010
regular expression searching ... 1011
regular expressions, developing ... 999
reindent-then-newline-and-indent ... 914
relative file name ... 639
relative remapping, faces ... 1180
remainder ... 47
remapping commands ... 504
remhash ... 129
remote-file-name-inhibit-cache ... 655
remote-file-name-inhibit-locks ... 621
remove ... 94
remove-from-invisibility-spec ... 1147
remove-function ... 255
remove-hook ... 527
remove-images ... 1228
remove-list-of-text-properties ... 922
remove-overlays ... 1155
remove-text-properties ... 922
remove-variable-watcher ... 200
removing from sequences ... 109
remq ... 93
rename-auto-save-file ... 669
rename-buffer ... 676
rename-file ... 632
rendering html ... 955
reordering, of bidirectional text ... 1255
reordering, of elements in lists ... 90
reparent frame ... 834
repeat events ... 447
repeat-mode ... 486
repeatable key bindings ... 486
repeated loading ... 309
replace bindings ... 500
replace characters ... 942
replace characters in region ... 942
replace characters in string ... 942
replace list element ... 88
replace matched text ... 1015
replace part of list ... 89
replace-buffer-contents ... 944
replace-buffer-in-windows ... 723
replace-match ... 1015
replace-re-search-function ... 1023
replace-regexp-in-region ... 1020
replace-regexp-in-string ... 1020
replace-region-contents ... 945
replace-search-function ... 1023
replace-string-in-region ... 1020
replacement after search ... 1019
replacing display specs ... 1204
require ... 311
require, customization keyword ... 279
require-final-newline ... 616
require-theme ... 297
requiring features ... 310
reserved keys ... 1339
reset, face attribute value ... 1168
resize window ... 705
resize-mini-frames ... 421
resize-mini-windows ... 420
rest arguments ... 234
restacking a frame ... 832
restart-emacs ... 1268
restore-buffer-modified-p ... 680
restricted-sexp, customization types ... 291
restriction (in a buffer) ... 867
resume (cf. no-redraw-on-reenter) ... 1133
resume-tty ... 1270
resume-tty-functions ... 1270
rethrow a signal ... 184
retrieve node, tree-sitter ... 1048
retrieving tree-sitter nodes ... 1049
return (ASCII character) ... 12
return value ... 230
reusable-frames, a buffer display action alist entry ... 733
reverse ... 104

reversing a list ... 105
reversing a string ... 105
reversing a vector ... 105
revert-buffer ... 669
revert-buffer-function ... 670
revert-buffer-in-progress-p ... 670
revert-buffer-insert-file-contents-function ... 670
revert-without-query ... 670
reverting buffers ... 669
rgb value ... 849
right dividers ... 1202
right-divider, prefix key ... 463
right-divider-width, a frame parameter ... 813
right-fringe, a frame parameter ... 813
right-fringe, prefix key ... 463
right-fringe-width ... 1194
right-margin, prefix key ... 463
right-margin-width ... 1210
right-to-left text ... 1255
ring data structure ... 123
ring-bell-function ... 1253
ring-copy ... 123
ring-elements ... 123
ring-empty-p ... 124
ring-insert ... 124
ring-insert-at-beginning ... 124
ring-length ... 123
ring-p ... 123
ring-ref ... 124
ring-remove ... 124
ring-resize ... 124
ring-size ... 123
risky, defcustom keyword ... 284
risky-local-variable-p ... 216
RLO ... 1258
rm ... 633
root window ... 694
root window of atomic window ... 759
round ... 45
rounding in conversions ... 43
rounding without conversion ... 48
rplaca ... 88
rplacd ... 88
run time stack ... 343
run-at-time ... 1287
run-hook-with-args ... 526
run-hook-with-args-until-failure ... 526
run-hook-with-args-until-success ... 526
run-hooks ... 526
run-mode-hooks ... 538
run-with-idle-timer ... 1289
run-with-timer ... 1288
running a hook when a window gets selected .. 698

S

S-expression ... 145
safe local variable ... 215
safe, defcustom keyword ... 284
safe-length ... 81
safe-local-eval-forms ... 216
safe-local-variable, property of variable ... 215
safe-local-variable-p ... 216
safe-local-variable-values ... 215
safe-magic (property) ... 654
safely encode a string ... 984
safely encode characters in a charset ... 984
safely encode region ... 983
safety of functions ... 266
same-window-buffer-names, replacement for .. 740
same-window-regexps, replacement for ... 740
save abbrevs in files ... 1071
save-abbrevs ... 1071
save-buffer ... 614
save-buffer-coding-system ... 982
save-current-buffer ... 675
save-excursion ... 866
save-mark-and-excursion ... 867
save-match-data ... 1019
save-restriction ... 868
save-selected-window ... 699
save-some-buffers ... 614
save-some-buffers-default-predicate ... 614
save-window-excursion ... 778
saved-face (face symbol property) ... 1174
SaveUnder feature ... 854
saving buffers ... 614
saving text properties ... 656
saving window information ... 777
scalable fonts ... 1185
scalable-fonts-allowed ... 1185
scan-lists ... 1034
scan-sexps ... 1034
scanning expressions ... 1033
scanning for character sets ... 978
scanning keymaps ... 508
scope ... 201
scoping rule ... 201
screen layout ... 28
screen lines, moving by ... 860
screen of terminal ... 692
screen refresh ... 1133
screen-gamma, a frame parameter ... 821
script symbols ... 975
scroll bar events, data in ... 460
scroll bars ... 1199
scroll-bar-background, a frame parameter ... 822
scroll-bar-event-ratio ... 460
scroll-bar-foreground, a frame parameter ... 822
scroll-bar-height ... 1202
scroll-bar-height, a frame parameter ... 812
scroll-bar-mode ... 1202
scroll-bar-scale ... 460

scroll-bar-width ... 1202
scroll-bar-width, a frame parameter ... 812
scroll-command property ... 768
scroll-conservatively ... 767
scroll-down ... 766
scroll-down-aggressively ... 767
scroll-down-command ... 766
scroll-error-top-bottom ... 768
scroll-left ... 771
scroll-margin ... 767
scroll-other-window ... 766
scroll-other-window-down ... 767
scroll-preserve-screen-position ... 768
scroll-right ... 771
scroll-step ... 768
scroll-up ... 766
scroll-up-aggressively ... 768
scroll-up-command ... 766
scrolling textually ... 765
search-backward ... 997
search-failed ... 996
search-forward ... 996
search-map ... 489
search-spaces-regexp ... 1014
searching ... 996
searching active keymaps for keys ... 491
searching and case ... 998
searching and replacing ... 1019
searching for overlays ... 1161
searching for regexp ... 1011
searching text properties ... 924
secondary selection ... 842
secondary storage ... 636
secure-hash ... 947
secure-hash-algorithms ... 947
security ... 1305
security vulnerabilities in text ... 948
security, and display specifications ... 1204
seed, for random number generation ... 52
select safe coding system ... 985
select window hooks ... 698
select-frame ... 828
select-frame-set-input-focus ... 828
select-safe-coding-system ... 985
select-safe-coding-system-accept-default-p ... 986
select-safe-coding-system-function ... 986
select-window ... 698
selected frame ... 827
selected window ... 698
selected-frame ... 827
selected-window ... 698
selected-window-group ... 700
selected-window-group-function ... 700
selecting a buffer ... 673
selecting a font ... 1184
selecting a window ... 698
selection (for window systems) ... 842
selection-coding-system ... 843
selective-display ... 1149
selective-display-ellipses ... 1150
selectively disabling font-lock fontifications ... 571
self-evaluating form ... 146
self-insert-and-exit ... 418
self-insert-command ... 886
self-insert-command override ... 501
self-insert-command, minor modes ... 547
self-insert-uses-region-functions ... 886
self-insertion ... 886
SELinux context ... 629
send-string-to-terminal ... 1292
sending signals ... 1100
sentence-end ... 1023, 1024
sentence-end-double-space ... 904
sentence-end-without-period ... 904
sentence-end-without-space ... 904
sentinel (of process) ... 1109
seq library ... 107
seq-concatenate ... 112
seq-contains-p ... 111
seq-count ... 111
seq-difference ... 113
seq-do ... 108
seq-doseq ... 114
seq-drop ... 107
seq-drop-while ... 108
seq-elt ... 107
seq-empty-p ... 111
seq-every-p ... 110
seq-filter ... 109
seq-find ... 110
seq-group-by ... 113
seq-intersection ... 113
seq-into ... 114
seq-keep ... 109
seq-length ... 107
seq-let ... 114
seq-map ... 108
seq-map-indexed ... 109
seq-mapcat ... 113
seq-mapn ... 109
seq-max ... 114
seq-min ... 114
seq-partition ... 113
seq-position ... 112
seq-positions ... 112
seq-random-elt ... 115
seq-reduce ... 110
seq-remove ... 109
seq-remove-at-position ... 109
seq-set-equal-p ... 111
seq-setq ... 115
seq-some ... 110
seq-sort ... 111
seq-sort-by ... 111
seq-split ... 108

seq-subseq ... 112
seq-take ... 108
seq-take-while ... 108
seq-union ... 113
seq-uniq ... 112
seqp ... 107
sequence ... 102
sequence destructuring ... 114, 115
sequence functions in seq ... 107
sequence iteration ... 114
sequence length ... 102
sequence maximum ... 114
sequence minimum ... 114
sequence reverse ... 104
sequencep ... 102
sequences, generalized ... 107
sequences, intersection of ... 113
sequences, union of ... 113
sequencing ... 158
sequencing pattern ... 165
sequential execution ... 158
serial connections ... 1125
serial-process-configure ... 1127
serial-term ... 1125
serializing ... 1128
server-after-make-frame-hook ... 789
session file ... 1296
session manager ... 1295
set ... 199
set, defcustom keyword ... 282
set-advertised-calling-convention ... 261
set-after, defcustom keyword ... 284
set-auto-coding ... 989
set-auto-mode ... 533
set-binary-mode ... 376
set-buffer ... 674
set-buffer-auto-saved ... 668
set-buffer-major-mode ... 533
set-buffer-modified-p ... 679
set-buffer-multibyte ... 970
set-case-syntax ... 76
set-case-syntax-delims ... 76
set-case-syntax-pair ... 76
set-case-table ... 75
set-category-table ... 1040
set-char-table-extra-slot ... 120
set-char-table-parent ... 120
set-char-table-range ... 121
set-charset-priority ... 976
set-coding-system-priority ... 991
set-default ... 213
set-default-file-modes ... 634
set-default-toplevel-value ... 213
set-display-table-slot ... 1249
set-face-attribute ... 1176
set-face-background ... 1177
set-face-bold ... 1177
set-face-extend ... 1177
set-face-font ... 1177
set-face-foreground ... 1177
set-face-inverse-video ... 1177
set-face-italic ... 1177
set-face-stipple ... 1177
set-face-underline ... 1177
set-file-acl ... 636
set-file-extended-attributes ... 635
set-file-modes ... 634
set-file-selinux-context ... 636
set-file-times ... 635
set-fontset-font ... 1187
set-frame-configuration ... 833
set-frame-font ... 799
set-frame-height ... 802
set-frame-parameter ... 805
set-frame-position ... 800
set-frame-selected-window ... 699
set-frame-size ... 802
set-frame-width ... 803
set-frame-window-state-change ... 785
set-fringe-bitmap-face ... 1198
set-input-method ... 994
set-input-mode ... 1290
set-keyboard-coding-system ... 993
set-keymap-parent ... 488
set-left-margin ... 905
set-mark ... 876
set-marker ... 875
set-marker-insertion-type ... 874
set-match-data ... 1019
set-message-function ... 1136
set-message-functions ... 1137
set-minibuffer-message ... 1137
set-minibuffer-window ... 420
set-mouse-absolute-pixel-position ... 839
set-mouse-pixel-position ... 839
set-mouse-position ... 838
set-multi-message ... 1137
set-network-process-option ... 1122
set-process-buffer ... 1103
set-process-coding-system ... 1098
set-process-datagram-address ... 1118
set-process-filter ... 1106
set-process-plist ... 1099
set-process-query-on-exit-flag ... 1111
set-process-sentinel ... 1110
set-process-thread ... 1108
set-process-window-size ... 1104
set-register ... 943
set-right-margin ... 905
set-standard-case-table ... 75
set-syntax-table ... 1031
set-terminal-coding-system ... 993
set-terminal-parameter ... 823
set-text-properties ... 922
set-transient-map ... 494
set-transient-map-timeout ... 495

set-visited-file-modtime ... 681
set-visited-file-name ... 678
set-window-buffer ... 722
set-window-combination-limit ... 718
set-window-configuration ... 777
set-window-dedicated-p ... 751
set-window-display-table ... 1250
set-window-fringes ... 1194
set-window-group-start ... 764
set-window-group-start-function ... 764
set-window-hscroll ... 772
set-window-margins ... 1210
set-window-next-buffers ... 749
set-window-parameter ... 779
set-window-point ... 761
set-window-prev-buffers ... 749
set-window-scroll-bars ... 1200
set-window-start ... 763
set-window-vscroll ... 770
set-xwidget-buffer ... 1234
set-xwidget-plist ... 1234
set-xwidget-query-on-exit-flag ... 1234
set_function_finalizer ... 1371
set_user_finalizer ... 1378
set_user_ptr ... 1378
setcar ... 88
setcdr ... 89
setenv ... 1271
setf ... 224
setopt ... 199
setplist ... 139
setq ... 198
setq-connection-local ... 221
setq-default ... 212
setq-local ... 209
sets ... 91
setting modes of files ... 631
setting-constant error ... 189
severity level ... 1143
sexp ... 145
sexp motion ... 863
sha1 ... 948
SHA hash ... 947, 949
shaded, a frame parameter ... 817
shadowed Lisp files ... 304
shadowing of variables ... 190
shared structure, read syntax ... 30
shell command arguments ... 1083
shell-command-history ... 395
shell-command-to-string ... 1089
shell-quote-argument ... 1083
shift-selection, and interactive spec ... 426
shift-translation ... 463
shortdoc-add-function ... 609
shorthands ... 142
should_quit ... 1379
show image ... 1227
show-help-function ... 934
shr-insert-document ... 955
shrink-window-if-larger-than-buffer ... 708
shy groups ... 1005
sibling window ... 695
side effect ... 145
side windows ... 754
side, a buffer display action alist entry ... 737
side-effect-free property ... 141
SIGHUP ... 1268
SIGINT ... 1268
signal ... 181
signal-process ... 1101
signal-process-functions ... 1102
signaling errors ... 180
signals ... 1100
SIGTERM ... 1268
SIGTSTP ... 1268
sigusr1 event ... 454
sigusr2 event ... 454
simple package ... 1309
sin ... 51
single file package ... 1309
single-function hook ... 525
single-key-description ... 603
sit-for ... 471
site-init.el ... 1352
site-lisp directories ... 303
site-load.el ... 1352
site-run-file ... 1264
site-start.el ... 1261
size of frame ... 793
size of image ... 1228
size of text on display ... 1161
size of window ... 701
skip-chars-backward ... 866
skip-chars-forward ... 865
skip-syntax-backward ... 1033
skip-syntax-forward ... 1033
skip-taskbar, a frame parameter ... 818
skipping characters ... 865
skipping characters of certain syntax ... 1033
skipping comments ... 1037
sleep-for ... 472
slice, image ... 1227
slot, a buffer display action alist entry ... 737
small-temporary-file-directory ... 645
smallest fixnum ... 39
smie-bnf->prec2 ... 584
smie-close-block ... 584
smie-config ... 591
smie-config-guess ... 591
smie-config-local ... 592
smie-config-save ... 591
smie-config-set-indent ... 592
smie-config-show-indent ... 592
smie-down-list ... 584
smie-merge-prec2s ... 584
smie-prec2->grammar ... 584

smie-precs->prec2 ... 584
smie-rule-bolp ... 589
smie-rule-hanging-p ... 589
smie-rule-next-p ... 589
smie-rule-parent ... 590
smie-rule-parent-p ... 589
smie-rule-prev-p ... 589
smie-rule-separator ... 590
smie-rule-sibling-p ... 590
smie-setup ... 583
SMIE ... 583
SMIE grammar ... 585
SMIE lexer ... 586
smooth-curveto ... 1223
smooth-quadratic-bezier-curveto ... 1223
snap-width, a frame parameter ... 816
Snarf-documentation ... 600
sort ... 106
sort-columns ... 911
sort-fields ... 911
sort-fold-case ... 910
sort-lines ... 911
sort-numeric-base ... 911
sort-numeric-fields ... 911
sort-pages ... 911
sort-paragraphs ... 911
sort-regexp-fields ... 910
sort-subr ... 908
sorting lists ... 106
sorting sequences ... 111
sorting text ... 908
sorting vectors ... 106
sound ... 1293
source breakpoints ... 352
space (ASCII character) ... 12
space display spec, and bidirectional text ... 1257
spaces, pixel specification ... 1206
spaces, specified height or width ... 1205
sparse keymap ... 482
SPC in minibuffer ... 391
special events ... 471
special form descriptions ... 4
special forms ... 149
special forms for control structures ... 158
special modes ... 532
special read syntax ... 9
special variables ... 205
special-event-map ... 494
special-form-p ... 149
special-mode ... 538
special-mode-map ... 1402
special-variable-p ... 205
Specified time is not representable ... 1276
specify coding system ... 989
specify color ... 848
speedups ... 1341
splicing (with backquote) ... 152
split-height-threshold ... 739
split-root-window-below ... 712
split-root-window-right ... 712
split-string ... 57
split-string-and-unquote ... 1084
split-string-default-separators ... 58
split-string-shell-command ... 1084
split-width-threshold ... 739
split-window ... 710
split-window, a window parameter ... 780
split-window-below ... 712
split-window-keep-point ... 712
split-window-preferred-function ... 738
split-window-right ... 712
split-window-sensibly ... 738
splitting windows ... 710
sqlite-available-p ... 952
sqlite-close ... 952
sqlite-columns ... 953
sqlite-commit ... 954
sqlite-execute ... 952
sqlite-finalize ... 954
sqlite-load-extension ... 954
sqlite-mode-open-file ... 954
sqlite-more-p ... 954
sqlite-next ... 953
sqlite-open ... 952
sqlite-pragma ... 954
sqlite-rollback ... 954
sqlite-select ... 953
sqlite-transaction ... 954
sqlite-version ... 954
sqlitep ... 952
sqrt ... 52
stable sort ... 106
stack allocated Lisp objects ... 1359
stack frame ... 339
stack overflow in regexp ... 1010
standalone-parent ... 594
standard abbrev tables ... 1074
standard colors for character terminals ... 820
standard error process ... 1091
standard errors ... 1395
standard hooks ... 1403
standard regexps used in editing ... 1023
standard syntax table ... 1025
standard-case-table ... 75
standard-category-table ... 1040
standard-display-table ... 1250
standard-input ... 375
standard-output ... 381
standard-syntax-table ... 1025
standard-translation-table-for-decode ... 979
standard-translation-table-for-encode ... 979
standards of coding style ... 1336
start-file-process ... 1093
start-file-process-shell-command ... 1094
start-process ... 1092

`start-process`, command-line arguments from minibuffer ... 1083
`start-process-shell-command` ... 1094
STARTTLS network connections ... 1115
startup of Emacs ... 1260
startup screen ... 1262
`startup-redirect-eln-cache` ... 331
`startup.el` ... 1260
statement object ... 953
`staticpro`, protection from GC ... 1364
`stderr` stream, use for debugging ... 378
sticky text properties ... 935
`sticky`, a frame parameter ... 817
stop points ... 346
`stop-process` ... 1101
stopbits, in serial connections ... 1127
stopping an infinite loop ... 337
stopping on events ... 352
storage of vector-like Lisp objects ... 1355
`store-match-data` ... 1019
`store-substring` ... 60
stream (for printing) ... 376
stream (for reading) ... 372
strike-through text ... 1170
`string` ... 55
string creation ... 55
string equality ... 60
string in keymap ... 495
string input stream ... 373
string length ... 102
string modification ... 59
string predicates ... 55
string reverse ... 104
string search ... 996
string to number ... 65
string to object ... 375
string to vector ... 114
string, number of bytes ... 968
string, writing a doc string ... 597
`string-as-multibyte` ... 970
`string-as-unibyte` ... 970
`string-bytes` ... 968
`string-chars-consed` ... 1360
`string-chop-newline` ... 59
`string-clean-whitespace` ... 58
`string-collate-equalp` ... 61
`string-collate-lessp` ... 62
`string-distance` ... 64
`string-equal` ... 61
`string-equal-ignore-case` ... 61
`string-fill` ... 59
`string-glyph-split` ... 1165
`string-greaterp` ... 62
`string-lessp` ... 62
`string-limit` ... 59
`string-lines` ... 59
`string-match` ... 1012
`string-match-p` ... 1013
`string-or-null-p` ... 55
`string-pad` ... 59
`string-pixel-width` ... 1165
`string-prefix-p` ... 63
`string-replace` ... 1020
`string-search` ... 63
`string-suffix-p` ... 63
`string-to-char` ... 66
`string-to-int` ... 65
`string-to-multibyte` ... 969
`string-to-number` ... 65
`string-to-syntax` ... 1038
`string-to-unibyte` ... 969
`string-trim` ... 59
`string-trim-left` ... 59
`string-trim-right` ... 59
`string-version-lessp` ... 63
`string-width` ... 1162
`string<` ... 61
`string=` ... 60
`stringp` ... 55
strings ... 54
strings with keyboard events ... 460
strings, formatting them ... 66
`strings-consed` ... 1360
structural matching ... 171
sub-sequence ... 112
submenu ... 516
subprocess ... 1081
subr ... 230
`subr-arity` ... 232
`subrp` ... 232
`subst-char-in-region` ... 942
`subst-char-in-string` ... 942
substitute characters ... 942
`substitute-command-keys` ... 601
`substitute-in-file-name` ... 642
`substitute-key-definition` ... 500, 503
`substitute-quotes` ... 602
substituting keys in documentation ... 600
`substring` ... 56
`substring-no-properties` ... 56
subtype of char-table ... 119
success handler ... 184
suggestions ... 1
super characters ... 14
support for touchscreens ... 449
`suppress-keymap` ... 501
surrogate minibuffer frame ... 826
suspend (cf. `no-redraw-on-reenter`) ... 1133
suspend evaluation ... 477
suspend major mode temporarily ... 527
`suspend-emacs` ... 1269
`suspend-frame` ... 1270
`suspend-hook` ... 1269
`suspend-resume-hook` ... 1269
`suspend-tty` ... 1269
`suspend-tty-functions` ... 1269

suspending Emacs 1268
suspicious text 948
suspicious text strings 948
svg path commands 1222
svg-circle 1219
svg-clip-path 1221
svg-create 1218
svg-ellipse 1219
svg-embed 1220
svg-embed-base-uri-image 1221
svg-gradient 1219
svg-image 1221
svg-line 1219
svg-node 1221
svg-path 1220
svg-polygon 1220
svg-polyline 1220
svg-rectangle 1219
svg-remove 1221
svg-text 1220
SVG images 1218
SVG object 1219
swap text between buffers 690
switch-to-buffer 724
switch-to-buffer-in-dedicated-window 724
switch-to-buffer-obey-display-actions 725
switch-to-buffer-other-frame 725
switch-to-buffer-other-window 725
switch-to-buffer-preserve-window-point 724
switch-to-next-buffer 750
switch-to-prev-buffer 749
switch-to-prev-buffer-skip 750
switch-to-prev-buffer-skip-regexp 751
switches on command line 1266
switching to a buffer 723
sxhash-eq 131
sxhash-eql 131
sxhash-equal 130
symbol 133
symbol components 133
symbol equality 135
symbol evaluation 146
symbol forms 146
symbol function indirection 147
symbol in keymap 496
symbol name hashing 135
symbol property 138
symbol that evaluates to itself 189
symbol with constant value 189
symbol with position 143
symbol, where defined 312
symbol-file 312
symbol-function 248
symbol-name 136
symbol-plist 139
symbol-value 197
symbol-with-pos-p 144
symbol-with-pos-pos 144
symbolic links 623
symbolic shorthands 142
symbolp 133
symbols-consed 1360
symbols-with-pos-enabled 144
symmetric cipher 949
synchronized multisession variables 228
synchronous subprocess 1084
syntactic font lock 577
syntax class 1026
syntax class table 1026
syntax code 1037
syntax descriptor 1026
syntax entry, setting 1030
syntax error (Edebug) 364
syntax flags 1029
syntax for characters 11
syntax highlighting and coloring 564
syntax of regular expressions 999
syntax table 1025
syntax table example 543
syntax table internals 1037
syntax tables (accessing elements of) 1038
syntax tables in modes 530
syntax tree nodes, by field name 1051
syntax tree nodes, by kinship 1050
syntax tree nodes, by position 1051
syntax tree nodes, retrieving from other nodes 1050
syntax tree, concrete 1043
syntax tree, from parsing program source 1042
syntax tree, retrieving nodes 1049
syntax trees, node information 1054
syntax-after 1038
syntax-class 1038
syntax-class-to-char 1038
syntax-ppss 1035
syntax-ppss-context 1036
syntax-ppss-flush-cache 1035
syntax-ppss-toplevel-pos 1036
syntax-propertize-extend-region-functions 1033
syntax-propertize-function 1032
syntax-table 1031
syntax-table (text property) 1032
syntax-table-p 1025
system abbrev 1069
system processes 1111
system tooltips 1254
system type and name 1270
system-configuration 1270
system-groups 1275
system-key-alist 1294
system-messages-locale 995
system-name 1271
system-time-locale 995
system-type 1270
system-users 1275

T

`t` ... 2
`t` input stream ... 373
`t` output stream ... 376
tab (ASCII character) ... 12
tab bar ... 796
tab deletion ... 889
`tab-always-indent` ... 914
tab-bar mouse events ... 446
`tab-bar`, prefix key ... 463
`tab-bar-lines`, a frame parameter ... 813
tab-bar-map ... 1402
`tab-first-completion` ... 914
`tab-line`, prefix key ... 463
`tab-line-format`, a window parameter ... 781
`tab-prefix-map` ... 489
`tab-stop-list` ... 917
`tab-to-tab-stop` ... 917
`tab-width` ... 1248
`TAB` in minibuffer ... 391
tabs stops for indentation ... 917
Tabulated List mode ... 539
`tabulated-list-clear-all-tags` ... 542
`tabulated-list-delete-entry` ... 541
`tabulated-list-entries` ... 540
`tabulated-list-format` ... 540
`tabulated-list-get-entry` ... 542
`tabulated-list-get-id` ... 542
`tabulated-list-gui-sort-indicator-asc` ... 539
`tabulated-list-gui-sort-indicator-desc` ... 540
`tabulated-list-header-overlay-p` ... 542
`tabulated-list-init-header` ... 541
`tabulated-list-mode` ... 539
`tabulated-list-padding` ... 542
`tabulated-list-print` ... 541
`tabulated-list-printer` ... 541
`tabulated-list-put-tag` ... 542
`tabulated-list-revert-hook` ... 541
`tabulated-list-set-col` ... 542
`tabulated-list-sort-key` ... 541
`tabulated-list-tty-sort-indicator-asc` ... 540
`tabulated-list-tty-sort-indicator-desc` ... 540
`tabulated-list-use-header-line` ... 542
tag on run time stack ... 178
`tag`, customization keyword ... 278
`take` ... 80
`tan` ... 51
TCP ... 1114
`temacs` ... 1351
`temp-buffer-max-height` ... 1153
`temp-buffer-max-width` ... 1153
`temp-buffer-resize-mode` ... 1152
`temp-buffer-setup-hook` ... 1152
`temp-buffer-show-function` ... 1151
`temp-buffer-show-hook` ... 1152
`temp-buffer-window-setup-hook` ... 1152
`temp-buffer-window-show-hook` ... 1152
`TEMP` environment variable ... 644
temporary buffer display ... 1150
temporary display ... 1150
temporary file on a remote host ... 645
temporary files ... 644
`temporary-file-directory` ... 644, 645
`term-file-aliases` ... 1265
`term-file-prefix` ... 1265
`TERM` environment variable ... 1265
Termcap ... 1265
terminal ... 787
terminal input ... 1290
terminal input modes ... 1290
terminal output ... 1292
terminal parameters ... 823
terminal screen ... 692
terminal type ... 27
`terminal-coding-system` ... 993
`terminal-list` ... 789
`terminal-live-p` ... 787
terminal-local variables ... 790
`terminal-name` ... 789
`terminal-parameter` ... 823
`terminal-parameters` ... 823
terminal-specific initialization ... 1265
terminology, for tree-sitter functions ... 1048
termscript file ... 1292
`terpri` ... 380
`test-completion` ... 398
`testcover-mark-all` ... 370
`testcover-next-mark` ... 370
`testcover-start` ... 370
testing types ... 31
text ... 880
text area ... 797
text area of a window ... 693
text comparison ... 60
text conversion of coding system ... 983
text deletion ... 888
text height of a frame ... 801
text insertion ... 884
text near point ... 880
text parsing ... 1025
text properties ... 919
text properties in files ... 656
text properties in the mode line ... 559
text properties, changing ... 921
text properties, examining ... 920
text properties, read syntax ... 21
text properties, searching ... 924
text representation ... 966
text size of a frame ... 801
text terminal ... 787
text width of a frame ... 801
`text-char-description` ... 604
`text-mode` ... 537
`text-mode-abbrev-table` ... 1074
`text-properties-at` ... 921
`text-property-any` ... 925

`text-property-default-nonsticky` ... 935
`text-property-not-all` ... 925
`text-property-search-backward` ... 927
`text-property-search-forward` ... 926
`text-quoting-style` ... 602, 603
text-terminal focus notification ... 828
`textsec-check` ... 948
`textsec-suspicious` (face) ... 949
`textsec-suspicious-p` ... 949
textual order ... 158
textual scrolling ... 765
`theme-face` (face symbol property) ... 1174
`thing-at-point` ... 883
`thing-at-point-provider-alist` ... 883
`this-command` ... 437
`this-command-keys` ... 438
`this-command-keys-shift-translated` ... 463
`this-command-keys-vector` ... 438
`this-original-command` ... 438
thread backtrace ... 1079
thread list ... 1079
`thread--blocker` ... 1077
`thread-join` ... 1076
`thread-last-error` ... 1077
`thread-list-refresh-seconds` ... 1079
`thread-live-p` ... 1077
`thread-name` ... 1077
`thread-signal` ... 1077
`thread-yield` ... 1077
`threadp` ... 1076
threads ... 1076
`three-step-help` ... 607
`throw` ... 178
`throw` example ... 476
thunk ... 155
`thunk-delay` ... 155
`thunk-force` ... 155
`thunk-let` ... 155
`thunk-let*` ... 156
tiled windows ... 692
time calculations ... 1285
time conversion ... 1278
time formatting ... 1281
time of day ... 1275
time parsing ... 1281
time value ... 1276
time zone rule ... 1277
time zone rules ... 1277
time zone, current ... 1277
`time-add` ... 1286
`time-convert` ... 1278
`time-equal-p` ... 1285
`time-less-p` ... 1285
`time-subtract` ... 1285
`time-to-day-in-year` ... 1286
`time-to-days` ... 1286
`timer-max-repeats` ... 1288
`timerp` ... 1286
timers ... 1286
timestamp of a mouse event ... 459
timestamp, Lisp ... 1275
timing programs ... 371
tips for documentation strings ... 1342
tips for faster Lisp code ... 1341
tips for writing Lisp ... 1336
title bar ... 795
`title`, a frame parameter ... 807
TLS network connections ... 1115
`TMP` environment variable ... 644
`TMPDIR` environment variable ... 644
`toggle-enable-multibyte-characters` ... 968
tool bar ... 518
tool-bar mouse events ... 446
`tool-bar-add-item` ... 520
`tool-bar-add-item-from-menu` ... 520
`tool-bar-border` ... 521
`tool-bar-button-margin` ... 521
`tool-bar-button-relief` ... 521
`tool-bar-lines`, a frame parameter ... 813
`tool-bar-local-item-from-menu` ... 521
`tool-bar-map` ... 520
`tool-bar-position`, a frame parameter ... 813
`tooltip` face ... 1254
tooltip for help strings ... 928
tooltip frames ... 1254
tooltip window ... 694
`tooltip-event-buffer` ... 1254
`tooltip-frame-parameters` ... 1254
`tooltip-functions` ... 1254
`tooltip-help-tips` ... 1254
`tooltip-mode` ... 1254
tooltips ... 1254
top and bottom window decorations ... 693
top frame ... 833
top position ratio ... 808
`top`, a frame parameter ... 809
`top-level` ... 477
top-level default value ... 213
top-level form ... 299
top-level frame ... 788
`top-visible`, a frame parameter ... 816
total height of a window ... 701
total pixel height of a window ... 702
total pixel width of a window ... 702
total width of a window ... 701
touch point, in touchscreen events ... 449
`touch-end` event ... 452
touchscreen events ... 449
`touchscreen-begin` event ... 449
`touchscreen-end` event ... 449
`touchscreen-update` event ... 449
`tq-close` ... 1114
`tq-create` ... 1113
`tq-enqueue` ... 1113
trace buffer ... 356
tracing Lisp programs ... 334

`track-mouse` ... 837
tracking frame size changes ... 803
trailing blanks in file names ... 621
trampolines, in native compilation ... 332
transaction queue ... 1113
transcendental functions ... 51
transient keymap ... 494
`transient-mark-mode` ... 877
`translate-region` ... 942
`translate-upper-case-key-bindings` ... 463
translating input events ... 467
translation keymap ... 505
translation tables ... 978
`translation-table-for-input` ... 979
transparency, frame ... 821
`transpose-regions` ... 944
trash ... 633, 651
tray notifications, MS-Windows ... 1300
tree-sitter extra node ... 1054
tree-sitter fontifications, overview ... 580
tree-sitter host and embedded languages ... 1062
tree-sitter missing node ... 1054
tree-sitter narrowing ... 1047
tree-sitter node field name ... 1044
tree-sitter node that has error ... 1055
tree-sitter nodes, comparing ... 1054
tree-sitter outdated node ... 1054
tree-sitter parse string ... 1048
tree-sitter parse tree, leaf node ... 1049
tree-sitter parse-tree, update and after-change callback ... 1048
tree-sitter parser, creating ... 1047
tree-sitter parser, using ... 1046
tree-sitter patterns as strings ... 1059
tree-sitter query pattern syntax ... 1056
tree-sitter, find node ... 1048
tree-sitter, live parsing node ... 1055
`treesit-available-p` ... 1042
`treesit-beginning-of-defun` ... 865
`treesit-buffer-root-node` ... 1050
`treesit-buffer-too-large` ... 1047
`treesit-check-indent` ... 595
`treesit-defun-at-point` ... 1065
`treesit-defun-name` ... 1065
`treesit-defun-name-function` ... 1066
`treesit-defun-tactic` ... 865
`treesit-defun-type-regexp` ... 865
`treesit-end-of-defun` ... 865
`treesit-explore-mode` ... 1044
`treesit-extra-load-path` ... 1042
`treesit-filter-child` ... 1053
`treesit-font-lock-feature-list` ... 581
`treesit-font-lock-recompute-features` ... 582
`treesit-font-lock-rules` ... 580
`treesit-font-lock-settings` ... 582
`treesit-fontify-with-override` ... 581
`treesit-indent-function` ... 592
`treesit-induce-sparse-tree` ... 1052
`treesit-inspect-mode` ... 1044
`treesit-language-abi-version` ... 1043
`treesit-language-at` ... 1062
`treesit-language-at-point-function` ... 1064
`treesit-language-available-p` ... 1043
`treesit-library-abi-version` ... 1043
`treesit-load-language-error` ... 1042
`treesit-load-name-override-list` ... 1043
`treesit-major-mode-setup` ... 1065
`treesit-max-buffer-size` ... 1046
`treesit-node-at` ... 1049
`treesit-node-buffer` ... 1054
`treesit-node-check` ... 1055
`treesit-node-child` ... 1050
`treesit-node-child-by-field-name` ... 1051
`treesit-node-child-count` ... 1055
`treesit-node-children` ... 1050
`treesit-node-descendant-for-range` ... 1051
`treesit-node-end` ... 1054
`treesit-node-eq` ... 1054
`treesit-node-field-name` ... 1055
`treesit-node-field-name-for-child` ... 1055
`treesit-node-first-child-for-pos` ... 1051
`treesit-node-index` ... 1055
`treesit-node-language` ... 1054
`treesit-node-next-sibling` ... 1050
`treesit-node-on` ... 1049
`treesit-node-outdated` ... 1049
`treesit-node-p` ... 1054
`treesit-node-parent` ... 1050
`treesit-node-parser` ... 1054
`treesit-node-prev-sibling` ... 1050
`treesit-node-start` ... 1054
`treesit-node-text` ... 1054
`treesit-node-top-level` ... 1053
`treesit-node-type` ... 1055
`treesit-parent-until` ... 1053
`treesit-parent-while` ... 1053
`treesit-parse-string` ... 1048
`treesit-parser-add-notifier` ... 1048
`treesit-parser-buffer` ... 1047
`treesit-parser-create` ... 1047
`treesit-parser-delete` ... 1047
`treesit-parser-included-ranges` ... 1061
`treesit-parser-language` ... 1047
`treesit-parser-list` ... 1047
`treesit-parser-notifiers` ... 1048
`treesit-parser-p` ... 1047
`treesit-parser-remove-notifier` ... 1048
`treesit-parser-root-node` ... 1050
`treesit-parser-set-included-ranges` ... 1061
`treesit-pattern-expand` ... 1060
`treesit-query-capture` ... 1056
`treesit-query-compile` ... 1060
`treesit-query-error` ... 1056
`treesit-query-expand` ... 1060
`treesit-query-language` ... 1060
`treesit-query-range` ... 1061

`treesit-query-string` ... 1057
`treesit-query-validate` ... 1056
`treesit-range-invalid` ... 1061
`treesit-range-rules` ... 1063
`treesit-range-settings` ... 1064
`treesit-ready-p` ... 1065
`treesit-search-forward` ... 1051
`treesit-search-forward-goto` ... 1052
`treesit-search-subtree` ... 1051
`treesit-simple-imenu-settings` ... 564
`treesit-simple-indent-presets` ... 593
`treesit-simple-indent-rules` ... 592
`treesit-update-ranges` ... 1062
triple-click events ... 447
true ... 2
true list ... 77
truename (of file) ... 625
`truncate` ... 44
`truncate-lines` ... 1134
`truncate-partial-width-windows` ... 1134
`truncate-string-ellipsis` ... 1163
`truncate-string-to-width` ... 1162
truth value ... 2
`try-completion` ... 396
`tty-color-alist` ... 851
`tty-color-approximate` ... 851
`tty-color-clear` ... 851
`tty-color-define` ... 850
`tty-color-mode`, a frame parameter ... 820
`tty-color-translate` ... 851
`tty-erase-char` ... 1273
`tty-menu-calls-mouse-position-function` ... 838
`tty-setup-hook` ... 1266
`tty-top-frame` ... 833
turn multibyte support on or off ... 968
two's complement ... 38
type ... 8
`type` (button property) ... 1237
type checking ... 31
type checking internals ... 1363
type of node, tree-sitter ... 1044
type predicates ... 31
`type`, `defcustom` keyword ... 285
`type-error`, customization keyword ... 294
`type-of` ... 33
`type_of` ... 1379
typographic conventions ... 2
`TZ`, environment variable ... 1277

U

UBA ... 1255
UDP ... 1114
UID ... 1274
umask ... 634
unassigned character codepoints ... 972
unbalanced parentheses ... 369
unbinding keys ... 508
`unbury-buffer` ... 685
`undecided` coding system ... 980
undecided coding-system, when decoding ... 992
`undecided` coding-system, when encoding ... 991
`undecorated`, a frame parameter ... 818
`undefined` ... 497
`undefined` in keymap ... 496
undefined key ... 482
`underline-minimum-offset` ... 1172
underlined text ... 1169
undo avoidance ... 942
undo in temporary buffers ... 675
`undo-amalgamate-change-group` ... 963
`undo-ask-before-discard` ... 901
`undo-auto-amalgamate` ... 899
`undo-auto-current-boundary-timer` ... 900
`undo-boundary` ... 899
`undo-in-progress` ... 900
`undo-limit` ... 901
`undo-outer-limit` ... 901
`undo-strong-limit` ... 901
unexec ... 1351, 1353
`unhandled-file-name-directory` ... 655
unibyte buffers, and bidi reordering ... 1255
unibyte text ... 966
`unibyte-char-to-multibyte` ... 969
`unibyte-string` ... 968
Unicode ... 966
unicode bidirectional algorithm ... 1255
unicode character escape ... 12
unicode general category ... 972
`unicode`, a charset ... 976
`unicode-category-table` ... 975
`unintern` ... 138
uninterned symbol ... 135
uninterned symbol, and generating Lisp code ... 136
union of sequences ... 113
unique file names ... 644
Universal Time ... 1275
`universal-argument` ... 475
`universal-argument-map` ... 1402
`unless` ... 160
`unload-feature` ... 314
`unload-feature-special-hooks` ... 314
unloading packages ... 314
unloading packages, preparing for ... 1337
`unlock-buffer` ... 620
unmapped frame ... 831
unnumbered group ... 1005
unpacking ... 1128
`unread-command-events` ... 469
`unsafep` ... 267
unspecified, face attribute value ... 1168
`unsplittable`, a frame parameter ... 814
unused lexical variable ... 206
`unwind-protect` ... 187
unwinding ... 187
`up-list` ... 863

`upcase` ... 73
`upcase-initials` ... 74
`upcase-region` ... 919
`upcase-word` ... 919
update callback, for tree-sitter parse-tree ... 1048
upper case ... 73
upper case key sequence ... 463
uptime of Emacs ... 1285
use time of window ... 700
`use-empty-active-region` ... 879
`use-frame-synchronization`, a frame parameter ... 817
`use-global-map` ... 493
`use-hard-newlines` ... 905
`use-local-map` ... 493
`use-package` ... 312
`use-region-p` ... 879
`use-system-tooltips` ... 1254
user errors, signaling ... 181
user groups ... 1275
user identification ... 1274
user options, how to define ... 281
user pointer object ... 316
user pointer, using in module functions ... 1377
user signals ... 454
user-defined error ... 186
`user-emacs-directory` ... 1265
`user-error` ... 181
`user-full-name` ... 1274
`user-init-file` ... 1264
`user-login-name` ... 1274
`user-mail-address` ... 1274
`user-position`, a frame parameter ... 809
user-ptr object ... 316
`user-ptrp` ... 316
`user-real-login-name` ... 1274
`user-real-uid` ... 1275
`user-size`, a frame parameter ... 810
`user-uid` ... 1275
UTC ... 1275
`utf-8-emacs` coding system ... 981
utility functions for parser-based indentation ... 595

V

valid windows ... 692
validity of coding system ... 983
value cell ... 133
value of expression ... 145
value of function ... 230
`values` ... 154
variable ... 189
variable aliases ... 222
variable definition ... 194
variable descriptions ... 6
variable pitch tables ... 539
variable watchpoints ... 200
variable with constant value ... 189
variable write debugging ... 338
variable, buffer-local ... 207
`variable-documentation` property ... 597
variable-width spaces ... 1205
variant coding system ... 980
`vc-mode` ... 555
`vc-prefix-map` ... 489
`vc-responsible-backend` ... 626
`vconcat` ... 118
`vec_get` ... 1374
`vec_set` ... 1374
`vec_size` ... 1374
`vector` ... 118
vector (type) ... 117
vector evaluation ... 146
vector length ... 102
vector reverse ... 104
vector to list ... 114
`vector-cells-consed` ... 1360
vector-like objects, storage ... 1355
`vectorp` ... 118
`verify-visited-file-modtime` ... 681
version number (in file name) ... 637
`version`, customization keyword ... 280
`version-control` ... 664
vertical combination ... 695
vertical fractional scrolling ... 769
vertical scroll position ... 769
vertical tab ... 12
`vertical-line`, prefix key ... 463
`vertical-lineto` ... 1222
`vertical-motion` ... 861
`vertical-scroll-bar` ... 1201
`vertical-scroll-bar`, prefix key ... 463
`vertical-scroll-bars`, a frame parameter ... 812
view part, model/view/controller ... 1241
`view-register` ... 943
virtual buffers ... 690
`visibility`, a frame parameter ... 817
visible frame ... 831
`visible-bell` ... 1253
`visible-frame-list` ... 825
visited file ... 677
visited file mode ... 533
`visited-file-modtime` ... 681
visiting files ... 610
visiting files, functions for ... 610
visual lines, moving by ... 860
visual order ... 1255
visual order, preserve when copying bidirectional text ... 1259
visual-order cursor motion ... 1257
void function ... 147
void function cell ... 249
void variable ... 193
`void-function` ... 248
`void-text-area-pointer` ... 842
`void-variable` error ... 193

W

`w32-collate-ignore-punctuation` ... 61
`w32-notification-close` ... 1301
`w32-notification-notify` ... 1300
`wait-for-wm`, a frame parameter ... 817
waiting ... 471
waiting for command key input ... 470
`waiting-for-user-input-p` ... 1110
`walk-windows` ... 720
`warn` ... 1143
warning options ... 1145
warning type ... 1143
warning variables ... 1144
`warning-fill-column` ... 1145
`warning-fill-prefix` ... 1144
`warning-levels` ... 1144
`warning-minimum-level` ... 1145
`warning-minimum-log-level` ... 1145
`warning-prefix-function` ... 1144
`warning-series` ... 1144
`warning-suppress-log-types` ... 1145
`warning-suppress-types` ... 1145
`warning-type-format` ... 1145
warnings ... 1142
warnings from byte compiler ... 1342
warnings, delayed ... 1146
warnings, suppressing during startup ... 1145
watch, for filesystem events ... 1301
watchpoints for Lisp variables ... 200
webkit browser widget ... 1233
`wheel-down` event ... 452
`wheel-up` event ... 452
`when` ... 160
`when-let` ... 161
where was a symbol defined ... 312
`where-is-internal` ... 509
`while` ... 174
`while-let` ... 161
`while-no-input` ... 470
`while-no-input-ignore-events` ... 470
whitespace ... 12
`wholenump` ... 42
`widen` ... 868
widening ... 868
width of a window ... 701
`width`, a frame parameter ... 810
window ... 692
`window` (overlay property) ... 1158
window auto-selection ... 776
window body ... 693
window body height ... 702
window body size ... 703
window body width ... 703
window buffer change ... 783
window change functions ... 782
window combination ... 695
window combination limit ... 717
window configuration (Edebug) ... 358
window configuration change ... 784
window configurations ... 777
window decorations ... 693
window dividers ... 1202
window end position ... 762
window excursions ... 867
window header line ... 560
window height ... 701
window history ... 748
window in direction ... 697
window internals ... 1387
window layout in a frame ... 28
window layout, all frames ... 28
window manager interaction, and frame parameters ... 816
window order by time of last use ... 700
window ordering, cyclic ... 719
window parameters ... 779
window pixel height ... 702
window pixel width ... 702
window point ... 761
window point internals ... 1389
window position ... 761, 772
window position on display ... 807
window positions and window managers ... 809
window resizing ... 705
window selected within a frame ... 698
window selection change ... 783
window size ... 701
window size change ... 783
window size on display ... 810
window size, changing ... 705
window splitting ... 710
window start position ... 762
window state ... 778
window state change ... 783
window state change flag ... 785
window that satisfies a predicate ... 721
window top line ... 762
window tree ... 694
window use time ... 700
window width ... 701
`window`, a buffer display action alist entry ... 737
`window-absolute-body-pixel-edges` ... 775
`window-absolute-pixel-edges` ... 774
`window-absolute-pixel-position` ... 775
`window-adjust-process-window-size-function` ... 1104
`window-at` ... 773
`window-at-side-p` ... 696
`window-atom`, a window parameter ... 781
`window-atom-root` ... 759
`window-body-edges` ... 773
`window-body-height` ... 702
`window-body-pixel-edges` ... 774
`window-body-size` ... 703
`window-body-width` ... 703
`window-bottom-divider-width` ... 1203

window-buffer ... 722
window-buffer-change-functions ... 783
window-bump-use-time ... 700
window-child ... 696
window-combination-limit ... 716, 718
window-combination-resize ... 718
window-combined-p ... 696
window-configuration-change-hook ... 784
window-configuration-equal-p ... 778
window-configuration-frame ... 778
window-configuration-p ... 778
window-current-scroll-bars ... 1201
window-dedicated-p ... 751
window-display-table ... 1250
window-edges ... 772
window-end ... 762
window-font-height ... 1193
window-font-width ... 1193
window-frame ... 694
window-fringes ... 1195
window-full-height-p ... 702
window-full-width-p ... 702
window-group-end ... 763
window-group-end-function ... 763
window-group-start ... 762
window-group-start-function ... 762
window-header-line-height ... 561, 704
window-height, a buffer display action alist entry ... 735
window-hscroll ... 771
window-id, a frame parameter ... 817
window-in-direction ... 697
window-largest-empty-rectangle ... 775
window-left-child ... 696
window-line-height ... 765
window-lines-pixel-dimensions ... 1164
window-list ... 694
window-live-p ... 692
window-main-window ... 756
window-make-atom ... 760
window-margins ... 1210
window-max-chars-per-line ... 704
window-min-height ... 704
window-min-height, a buffer display action alist entry ... 734
window-min-size ... 704
window-min-width ... 704
window-min-width, a buffer display action alist entry ... 734
window-minibuffer-p ... 420
window-mode-line-height ... 703
window-next-buffers ... 749
window-next-sibling ... 696
window-old-body-pixel-height ... 786
window-old-body-pixel-width ... 786
window-old-buffer ... 785
window-old-pixel-height ... 785
window-old-pixel-width ... 785
window-parameter ... 779
window-parameters ... 779
window-parameters, a buffer display action alist entry ... 734
window-parent ... 695
window-persistent-parameters ... 780
window-pixel-edges ... 774
window-pixel-height ... 702
window-pixel-width ... 702
window-point ... 761
window-point-insertion-type ... 762
window-preserve-size ... 709
window-preserved-size ... 710
window-preserved-size, a window parameter ... 781
window-prev-buffers ... 748
window-prev-sibling ... 696
window-resizable ... 705
window-resize ... 706
window-resize-pixelwise ... 706
window-right-divider-width ... 1203
window-scroll-bar-height ... 1201
window-scroll-bar-width ... 1201
window-scroll-bars ... 1201
window-scroll-functions ... 782
window-selection-change-functions ... 783
window-setup-hook ... 1264
window-side, a window parameter ... 781
window-sides-reversed ... 756
window-sides-slots ... 755
window-sides-vertical ... 755
window-size, a buffer display action alist entry ... 735
window-size-change-functions ... 783
window-size-fixed ... 709
window-slot, a window parameter ... 781
window-start ... 762
window-state-change-functions ... 784
window-state-change-hook ... 784
window-state-get ... 778
window-state-put ... 779
window-swap-states ... 779
window-system ... 1253
window-system window ... 692
window-system-initialization ... 1260
window-tab-line-height ... 704
window-text-pixel-size ... 1163
window-toggle-side-windows ... 756
window-top-child ... 696
window-total-height ... 701
window-total-size ... 702
window-total-width ... 701
window-tree ... 697
window-use-time ... 700
window-valid-p ... 693
window-vscroll ... 769
window-width, a buffer display action alist entry ... 735

windowp ... 692
windows, controlling precisely ... 722
windows, recombining ... 714
with-case-table ... 76
with-coding-priority ... 991
with-connection-local-application-variables ... 220
with-connection-local-variables ... 221
with-current-buffer ... 675
with-current-buffer-window ... 1152
with-delayed-message ... 1141
with-demoted-errors ... 186
with-environment-variables ... 1272
with-eval-after-load ... 315
with-existing-directory ... 622
with-file-modes ... 634
with-help-window ... 606
with-local-quit ... 473
with-mutex ... 1078
with-no-warnings ... 323
with-output-to-string ... 380
with-output-to-temp-buffer ... 1150
with-restriction ... 869
with-selected-frame ... 700
with-selected-window ... 700
with-silent-modifications ... 680
with-silent-modifications, and changes in text properties ... 923
with-sqlite-transaction ... 954
with-suppressed-warnings ... 323
with-syntax-table ... 1031
with-temp-buffer ... 675
with-temp-buffer-window ... 1152
with-temp-file ... 619
with-temp-message ... 1137
with-timeout ... 1288
with-undo-amalgamate ... 900
without-restriction ... 869
word-search-backward ... 998
word-search-backward-lax ... 998
word-search-forward ... 997
word-search-forward-lax ... 998
word-search-regexp ... 997
words in region ... 860
words-include-escapes ... 857
wrap-prefix ... 1135
write-abbrev-file ... 1072
write-char ... 380
write-contents-functions ... 615
write-file ... 614
write-file-functions ... 615
write-region ... 618, 636
write-region-annotate-functions ... 659
write-region-inhibit-fsync ... 619
write-region-post-annotation-function ... 660
writing a documentation string ... 597
writing buffer display actions ... 745
writing emacs modules ... 1366
writing Emacs primitives ... 1361
writing module functions ... 1368
writing to files ... 618
wrong-number-of-arguments ... 234
wrong-type-argument ... 31

X

x-alt-keysym ... 1294
x-alternatives-map ... 1402
x-begin-drag ... 847
x-bitmap-file-path ... 1172
x-close-connection ... 791
x-color-defined-p ... 849
x-color-values ... 850
x-ctrl-keysym ... 1294
x-defined-colors ... 849
x-display-color-p ... 852
x-display-list ... 791
x-dnd-direct-save-function ... 845
x-dnd-disable-motif-protocol ... 848
x-dnd-known-types ... 844
x-dnd-save-direct ... 845
x-dnd-save-direct-immediately ... 845
x-dnd-test-function ... 844
x-dnd-types-alist ... 844
x-dnd-use-offix-drop ... 848
x-dnd-use-unsupported-drop ... 848
x-double-buffered-p ... 832
x-family-fonts ... 1186
x-focus-frame ... 828
x-get-resource ... 851
x-hyper-keysym ... 1294
x-list-fonts ... 1185
x-meta-keysym ... 1294
x-open-connection ... 791
x-parse-geometry ... 822
x-pointer-shape ... 842
x-popup-dialog ... 841
x-popup-menu ... 839
x-pre-popup-menu-hook ... 840
x-resource-class ... 851
x-resource-name ... 851
x-sensitive-text-pointer-shape ... 842
x-server-vendor ... 854
x-server-version ... 854
x-setup-function-keys ... 1402
x-stretch-cursor ... 819
x-super-keysym ... 1294
X display names ... 790
X Window System ... 1253
X11 keysyms ... 1294
XBM ... 1216
xdigit character class, regexp ... 1003
XDS ... 845
XML DOM ... 956
xor ... 163
XPM ... 1217

xwidget 1233
xwidget callbacks 450
xwidget property list 1234
xwidget type 30
xwidget-buffer 1234
xwidget-display-event event 451
xwidget-event event 450
xwidget-info 1234
xwidget-live-p 1233
xwidget-perform-lispy-event 1235
xwidget-plist 1234
xwidget-query-on-exit-flag 1235
xwidget-resize 1234
xwidget-size-request 1234
xwidget-view type 30
xwidget-webkit-back-forward-list 1236
xwidget-webkit-estimated-load-progress 1236
xwidget-webkit-execute-script 1234
xwidget-webkit-execute-script-rv 1234
xwidget-webkit-finish-search 1235
xwidget-webkit-get-title 1234
xwidget-webkit-goto-history 1236
xwidget-webkit-goto-uri 1234
xwidget-webkit-load-html 1235
xwidget-webkit-next-result 1235
xwidget-webkit-previous-result 1235
xwidget-webkit-search 1235
xwidget-webkit-set-cookie-storage-file 1236
xwidget-webkit-stop-loading 1236
xwidgetp 1233

Y

y-or-n-p 414
y-or-n-p-with-timeout 415
yank 894
yank suppression 501
yank-excluded-properties 894
yank-handled-properties 894
yank-media-handler 843
yank-pop 895
yank-transform-functions 894
yank-undo-function 895
yanking and text properties 894
yes-or-no questions 414
yes-or-no-p 415

Z

z-group, a frame parameter 809
Z-order 832
zerop 42
zlib-available-p 945
zlib-decompress-region 945