

# **GNU Emacs Lisp Reference Manual**

---

For Emacs Version 22.1  
Revision 2.9, April 2007

---

by Bil Lewis, Dan LaLiberte, Richard Stallman  
and the GNU Manual Group

---

This is edition 2.9 of the GNU Emacs Lisp Reference Manual,  
corresponding to Emacs version 22.1.

Copyright © 1990, 1991, 1992, 1993, 1994, 1995, 1996, 1998, 1999, 2000, 2001, 2002, 2003,  
2004, 2005, 2006, 2007 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 1.2 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 freedom to copy and modify  
this GNU Manual, like GNU software. Copies published by the Free Software  
Foundation raise funds for GNU development.”

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 Suvasta.

## Short Contents

1	Introduction . . . . .	1
2	Lisp Data Types . . . . .	8
3	Numbers . . . . .	32
4	Strings and Characters . . . . .	47
5	Lists . . . . .	63
6	Sequences, Arrays, and Vectors . . . . .	87
7	Hash Tables . . . . .	97
8	Symbols . . . . .	102
9	Evaluation . . . . .	110
10	Control Structures . . . . .	119
11	Variables . . . . .	135
12	Functions . . . . .	160
13	Macros . . . . .	176
14	Writing Customization Definitions . . . . .	185
15	Loading . . . . .	201
16	Byte Compilation . . . . .	214
17	Advising Emacs Lisp Functions . . . . .	226
18	Debugging Lisp Programs . . . . .	237
19	Reading and Printing Lisp Objects . . . . .	268
20	Minibuffers . . . . .	278
21	Command Loop . . . . .	304
22	Keymaps . . . . .	347
23	Major and Minor Modes . . . . .	382
24	Documentation . . . . .	425
25	Files . . . . .	434
26	Backups and Auto-Saving . . . . .	471
27	Buffers . . . . .	481
28	Windows . . . . .	497
29	Frames . . . . .	529
30	Positions . . . . .	559
31	Markers . . . . .	572
32	Text . . . . .	581
33	Non-ASCII Characters . . . . .	640
34	Searching and Matching . . . . .	661
35	Syntax Tables . . . . .	684

36	Abbrevs and Abbrev Expansion . . . . .	699
37	Processes . . . . .	705
38	Emacs Display . . . . .	739
39	Operating System Interface . . . . .	812
A	Emacs 21 Antinews . . . . .	838
B	GNU Free Documentation License . . . . .	843
C	GNU General Public License . . . . .	850
D	Tips and Conventions . . . . .	856
E	GNU Emacs Internals . . . . .	870
F	Standard Errors . . . . .	891
G	Buffer-Local Variables . . . . .	895
H	Standard Keymaps . . . . .	899
I	Standard Hooks . . . . .	903
	Index . . . . .	908

# Table of Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Caveats	1
1.2	Lisp History	2
1.3	Conventions	2
1.3.1	Some Terms	2
1.3.2	<code>nil</code> and <code>t</code>	2
1.3.3	Evaluation Notation	3
1.3.4	Printing Notation	3
1.3.5	Error Messages	4
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	Acknowledgements	7
<b>2</b>	<b>Lisp Data Types</b>	<b>8</b>
2.1	Printed Representation and Read Syntax	8
2.2	Comments	9
2.3	Programming Types	9
2.3.1	Integer Type	9
2.3.2	Floating Point Type	10
2.3.3	Character Type	10
2.3.3.1	Basic Char Syntax	10
2.3.3.2	General Escape Syntax	11
2.3.3.3	Control-Character Syntax	12
2.3.3.4	Meta-Character Syntax	12
2.3.3.5	Other Character Modifier Bits	13
2.3.4	Symbol Type	13
2.3.5	Sequence Types	14
2.3.6	Cons Cell and List Types	14
2.3.6.1	Drawing Lists as Box Diagrams	15
2.3.6.2	Dotted Pair Notation	16
2.3.6.3	Association List Type	17
2.3.7	Array Type	18
2.3.8	String Type	18
2.3.8.1	Syntax for Strings	18
2.3.8.2	Non-ASCII Characters in Strings	19
2.3.8.3	Nonprinting Characters in Strings	19
2.3.8.4	Text Properties in Strings	20
2.3.9	Vector Type	20
2.3.10	Char-Table Type	20
2.3.11	Bool-Vector Type	21

2.3.12	Hash Table Type.....	21
2.3.13	Function Type.....	21
2.3.14	Macro Type.....	22
2.3.15	Primitive Function Type .....	22
2.3.16	Byte-Code Function Type .....	22
2.3.17	Autoload Type.....	23
2.4	Editing Types.....	23
2.4.1	Buffer Type.....	23
2.4.2	Marker Type.....	24
2.4.3	Window Type.....	24
2.4.4	Frame Type.....	25
2.4.5	Window Configuration Type.....	25
2.4.6	Frame Configuration Type .....	25
2.4.7	Process Type.....	25
2.4.8	Stream Type.....	25
2.4.9	Keymap Type.....	26
2.4.10	Overlay Type.....	26
2.5	Read Syntax for Circular Objects .....	26
2.6	Type Predicates.....	27
2.7	Equality Predicates .....	30
<b>3</b>	<b>Numbers .....</b>	<b>32</b>
3.1	Integer Basics.....	32
3.2	Floating Point Basics .....	33
3.3	Type Predicates for Numbers.....	34
3.4	Comparison of Numbers.....	35
3.5	Numeric Conversions.....	36
3.6	Arithmetic Operations .....	38
3.7	Rounding Operations .....	40
3.8	Bitwise Operations on Integers .....	41
3.9	Standard Mathematical Functions .....	44
3.10	Random Numbers.....	45
<b>4</b>	<b>Strings and Characters.....</b>	<b>47</b>
4.1	String and Character Basics.....	47
4.2	The Predicates for Strings.....	48
4.3	Creating Strings .....	48
4.4	Modifying Strings.....	51
4.5	Comparison of Characters and Strings .....	52
4.6	Conversion of Characters and Strings .....	54
4.7	Formatting Strings.....	56
4.8	Case Conversion in Lisp .....	58
4.9	The Case Table .....	60

<b>5 Lists . . . . .</b>	<b>63</b>
5.1 Lists and Cons Cells . . . . .	63
5.2 Predicates on Lists . . . . .	64
5.3 Accessing Elements of Lists . . . . .	64
5.4 Building Cons Cells and Lists . . . . .	67
5.5 Modifying List Variables . . . . .	71
5.6 Modifying Existing List Structure . . . . .	73
5.6.1 Altering List Elements with <code>setcar</code> . . . . .	73
5.6.2 Altering the CDR of a List . . . . .	74
5.6.3 Functions that Rearrange Lists . . . . .	75
5.7 Using Lists as Sets . . . . .	78
5.8 Association Lists . . . . .	81
5.9 Managing a Fixed-Size Ring of Objects . . . . .	84
<b>6 Sequences, Arrays, and Vectors . . . . .</b>	<b>87</b>
6.1 Sequences . . . . .	87
6.2 Arrays . . . . .	89
6.3 Functions that Operate on Arrays . . . . .	90
6.4 Vectors . . . . .	91
6.5 Functions for Vectors . . . . .	92
6.6 Char-Tables . . . . .	93
6.7 Bool-vectors . . . . .	95
<b>7 Hash Tables . . . . .</b>	<b>97</b>
7.1 Creating Hash Tables . . . . .	97
7.2 Hash Table Access . . . . .	99
7.3 Defining Hash Comparisons . . . . .	99
7.4 Other Hash Table Functions . . . . .	100
<b>8 Symbols . . . . .</b>	<b>102</b>
8.1 Symbol Components . . . . .	102
8.2 Defining Symbols . . . . .	103
8.3 Creating and Interning Symbols . . . . .	104
8.4 Property Lists . . . . .	107
8.4.1 Property Lists and Association Lists . . . . .	107
8.4.2 Property List Functions for Symbols . . . . .	108
8.4.3 Property Lists Outside Symbols . . . . .	108

<b>9 Evaluation . . . . .</b>	<b>110</b>
9.1 Kinds of Forms . . . . .	111
9.1.1 Self-Evaluating Forms . . . . .	111
9.1.2 Symbol Forms . . . . .	111
9.1.3 Classification of List Forms . . . . .	112
9.1.4 Symbol Function Indirection . . . . .	112
9.1.5 Evaluation of Function Forms . . . . .	113
9.1.6 Lisp Macro Evaluation . . . . .	113
9.1.7 Special Forms . . . . .	114
9.1.8 Autoloading . . . . .	115
9.2 Quoting . . . . .	115
9.3 Eval . . . . .	116
<b>10 Control Structures . . . . .</b>	<b>119</b>
10.1 Sequencing . . . . .	119
10.2 Conditionals . . . . .	120
10.3 Constructs for Combining Conditions . . . . .	122
10.4 Iteration . . . . .	124
10.5 Nonlocal Exits . . . . .	125
10.5.1 Explicit Nonlocal Exits: <code>catch</code> and <code>throw</code> . . . . .	125
10.5.2 Examples of <code>catch</code> and <code>throw</code> . . . . .	126
10.5.3 Errors . . . . .	127
10.5.3.1 How to Signal an Error . . . . .	128
10.5.3.2 How Emacs Processes Errors . . . . .	129
10.5.3.3 Writing Code to Handle Errors . . . . .	129
10.5.3.4 Error Symbols and Condition Names . . . . .	132
10.5.4 Cleaning Up from Nonlocal Exits . . . . .	133
<b>11 Variables . . . . .</b>	<b>135</b>
11.1 Global Variables . . . . .	135
11.2 Variables that Never Change . . . . .	135
11.3 Local Variables . . . . .	136
11.4 When a Variable is “Void” . . . . .	138
11.5 Defining Global Variables . . . . .	139
11.6 Tips for Defining Variables Robustly . . . . .	141
11.7 Accessing Variable Values . . . . .	143
11.8 How to Alter a Variable Value . . . . .	143
11.9 Scoping Rules for Variable Bindings . . . . .	145
11.9.1 Scope . . . . .	145
11.9.2 Extent . . . . .	146
11.9.3 Implementation of Dynamic Scoping . . . . .	146
11.9.4 Proper Use of Dynamic Scoping . . . . .	147
11.10 Buffer-Local Variables . . . . .	147
11.10.1 Introduction to Buffer-Local Variables . . . . .	148
11.10.2 Creating and Deleting Buffer-Local Bindings . . . . .	149
11.10.3 The Default Value of a Buffer-Local Variable . . . . .	152
11.11 Frame-Local Variables . . . . .	153

11.12 Possible Future Local Variables .....	155
11.13 File Local Variables.....	155
11.14 Variable Aliases .....	157
11.15 Variables with Restricted Values .....	158
<b>12 Functions .....</b>	<b>160</b>
12.1 What Is a Function? .....	160
12.2 Lambda Expressions .....	161
12.2.1 Components of a Lambda Expression.....	162
12.2.2 A Simple Lambda-Expression Example.....	162
12.2.3 Other Features of Argument Lists .....	163
12.2.4 Documentation Strings of Functions.....	164
12.3 Naming a Function .....	165
12.4 Defining Functions.....	165
12.5 Calling Functions .....	167
12.6 Mapping Functions .....	168
12.7 Anonymous Functions .....	170
12.8 Accessing Function Cell Contents .....	171
12.9 Declaring Functions Obsolete .....	173
12.10 Inline Functions .....	173
12.11 Determining whether a Function is Safe to Call .....	174
12.12 Other Topics Related to Functions .....	174
<b>13 Macros.....</b>	<b>176</b>
13.1 A Simple Example of a Macro.....	176
13.2 Expansion of a Macro Call.....	176
13.3 Macros and Byte Compilation.....	177
13.4 Defining Macros .....	178
13.5 Backquote .....	179
13.6 Common Problems Using Macros .....	180
13.6.1 Wrong Time .....	180
13.6.2 Evaluating Macro Arguments Repeatedly.....	180
13.6.3 Local Variables in Macro Expansions .....	182
13.6.4 Evaluating Macro Arguments in Expansion .....	182
13.6.5 How Many Times is the Macro Expanded? .....	183
13.7 Indenting Macros .....	184
<b>14 Writing Customization Definitions .....</b>	<b>185</b>
14.1 Common Item Keywords.....	185
14.2 Defining Customization Groups .....	187
14.3 Defining Customization Variables .....	188
14.4 Customization Types .....	191
14.4.1 Simple Types .....	191
14.4.2 Composite Types .....	194
14.4.3 Splicing into Lists.....	197
14.4.4 Type Keywords .....	198
14.4.5 Defining New Types .....	199

<b>15 Loading .....</b>	<b>201</b>
15.1 How Programs Do Loading .....	201
15.2 Load Suffixes .....	203
15.3 Library Search .....	203
15.4 Loading Non-ASCII Characters .....	205
15.5 Autoload .....	206
15.6 Repeated Loading .....	208
15.7 Features .....	209
15.8 Which File Defined a Certain Symbol .....	211
15.9 Unloading .....	211
15.10 Hooks for Loading .....	212
<b>16 Byte Compilation .....</b>	<b>214</b>
16.1 Performance of Byte-Compiled Code .....	214
16.2 The Compilation Functions .....	215
16.3 Documentation Strings and Compilation .....	217
16.4 Dynamic Loading of Individual Functions .....	218
16.5 Evaluation During Compilation .....	219
16.6 Compiler Errors .....	220
16.7 Byte-Code Function Objects .....	220
16.8 Disassembled Byte-Code .....	221
<b>17 Advising Emacs Lisp Functions .....</b>	<b>226</b>
17.1 A Simple Advice Example .....	226
17.2 Defining Advice .....	227
17.3 Around-Advice .....	229
17.4 Computed Advice .....	230
17.5 Activation of Advice .....	230
17.6 Enabling and Disabling Advice .....	232
17.7 Preactivation .....	232
17.8 Argument Access in Advice .....	233
17.9 Advising Primitives .....	234
17.10 The Combined Definition .....	235
<b>18 Debugging Lisp Programs .....</b>	<b>237</b>
18.1 The Lisp Debugger .....	237
18.1.1 Entering the Debugger on an Error .....	237
18.1.2 Debugging Infinite Loops .....	239
18.1.3 Entering the Debugger on a Function Call .....	239
18.1.4 Explicit Entry to the Debugger .....	240
18.1.5 Using the Debugger .....	240
18.1.6 Debugger Commands .....	241
18.1.7 Invoking the Debugger .....	242
18.1.8 Internals of the Debugger .....	243
18.2 Edebug .....	245
18.2.1 Using Edebug .....	246
18.2.2 Instrumenting for Edebug .....	247

18.2.3	Edebug Execution Modes . . . . .	247
18.2.4	Jumping . . . . .	249
18.2.5	Miscellaneous Edebug Commands . . . . .	249
18.2.6	Breaks . . . . .	250
18.2.6.1	Edebug Breakpoints . . . . .	250
18.2.6.2	Global Break Condition . . . . .	251
18.2.6.3	Source Breakpoints . . . . .	251
18.2.7	Trapping Errors . . . . .	252
18.2.8	Edebug Views . . . . .	252
18.2.9	Evaluation . . . . .	253
18.2.10	Evaluation List Buffer . . . . .	253
18.2.11	Printing in Edebug . . . . .	254
18.2.12	Trace Buffer . . . . .	255
18.2.13	Coverage Testing . . . . .	256
18.2.14	The Outside Context . . . . .	257
18.2.14.1	Checking Whether to Stop . . . . .	257
18.2.14.2	Edebug Display Update . . . . .	257
18.2.14.3	Edebug Recursive Edit . . . . .	258
18.2.15	Edebug and Macros . . . . .	258
18.2.15.1	Instrumenting Macro Calls . . . . .	258
18.2.15.2	Specification List . . . . .	259
18.2.15.3	Backtracking in Specifications . . . . .	262
18.2.15.4	Specification Examples . . . . .	263
18.2.16	Edebug Options . . . . .	263
18.3	Debugging Invalid Lisp Syntax . . . . .	265
18.3.1	Excess Open Parentheses . . . . .	265
18.3.2	Excess Close Parentheses . . . . .	266
18.4	Test Coverage . . . . .	266
18.5	Debugging Problems in Compilation . . . . .	267
<b>19</b>	<b>Reading and Printing Lisp Objects . . . . .</b>	<b>268</b>
19.1	Introduction to Reading and Printing . . . . .	268
19.2	Input Streams . . . . .	268
19.3	Input Functions . . . . .	271
19.4	Output Streams . . . . .	271
19.5	Output Functions . . . . .	273
19.6	Variables Affecting Output . . . . .	276
<b>20</b>	<b>Minibuffers . . . . .</b>	<b>278</b>
20.1	Introduction to Minibuffers . . . . .	278
20.2	Reading Text Strings with the Minibuffer . . . . .	279
20.3	Reading Lisp Objects with the Minibuffer . . . . .	281
20.4	Minibuffer History . . . . .	282
20.5	Initial Input . . . . .	284
20.6	Completion . . . . .	285
20.6.1	Basic Completion Functions . . . . .	285
20.6.2	Completion and the Minibuffer . . . . .	288
20.6.3	Minibuffer Commands that Do Completion . . . . .	289

20.6.4	High-Level Completion Functions .....	291
20.6.5	Reading File Names.....	293
20.6.6	Programmed Completion.....	296
20.7	Yes-or-No Queries .....	296
20.8	Asking Multiple Y-or-N Questions .....	298
20.9	Reading a Password .....	299
20.10	Minibuffer Commands .....	300
20.11	Minibuffer Windows .....	300
20.12	Minibuffer Contents .....	301
20.13	Recursive Minibuffers.....	302
20.14	Minibuffer Miscellany .....	302

## 21 Command Loop ..... 304

21.1	Command Loop Overview.....	304
21.2	Defining Commands .....	305
21.2.1	Using <code>interactive</code> .....	305
21.2.2	Code Characters for <code>interactive</code> .....	307
21.2.3	Examples of Using <code>interactive</code> .....	309
21.3	Interactive Call .....	310
21.4	Information from the Command Loop .....	312
21.5	Adjusting Point After Commands .....	315
21.6	Input Events .....	315
21.6.1	Keyboard Events .....	315
21.6.2	Function Keys .....	316
21.6.3	Mouse Events .....	317
21.6.4	Click Events .....	318
21.6.5	Drag Events .....	319
21.6.6	Button-Down Events.....	320
21.6.7	Repeat Events .....	320
21.6.8	Motion Events .....	321
21.6.9	Focus Events .....	322
21.6.10	Miscellaneous System Events.....	322
21.6.11	Event Examples .....	324
21.6.12	Classifying Events .....	324
21.6.13	Accessing Events.....	326
21.6.14	Putting Keyboard Events in Strings.....	328
21.7	Reading Input .....	329
21.7.1	Key Sequence Input.....	330
21.7.2	Reading One Event .....	331
21.7.3	Modifying and Translating Input Events.....	333
21.7.4	Invoking the Input Method.....	334
21.7.5	Quoted Character Input.....	335
21.7.6	Miscellaneous Event Input Features .....	335
21.8	Special Events .....	337
21.9	Waiting for Elapsed Time or Input .....	337
21.10	Quitting .....	338
21.11	Prefix Command Arguments .....	340
21.12	Recursive Editing.....	342

21.13	Disabling Commands .....	344
21.14	Command History .....	344
21.15	Keyboard Macros.....	345
<b>22</b>	<b>Keymaps.....</b>	<b>347</b>
22.1	Key Sequences.....	347
22.2	Keymap Basics .....	348
22.3	Format of Keymaps.....	348
22.4	Creating Keymaps.....	350
22.5	Inheritance and Keymaps.....	351
22.6	Prefix Keys.....	352
22.7	Active Keymaps .....	353
22.8	Searching the Active Keymaps .....	355
22.9	Controlling the Active Keymaps .....	356
22.10	Key Lookup .....	358
22.11	Functions for Key Lookup .....	360
22.12	Changing Key Bindings.....	361
22.13	Remapping Commands .....	364
22.14	Keymaps for Translating Sequences of Events .....	365
22.15	Commands for Binding Keys.....	367
22.16	Scanning Keymaps .....	368
22.17	Menu Keymaps.....	370
22.17.1	Defining Menus .....	370
22.17.1.1	Simple Menu Items .....	371
22.17.1.2	Extended Menu Items .....	372
22.17.1.3	Menu Separators.....	374
22.17.1.4	Alias Menu Items .....	375
22.17.2	Menus and the Mouse.....	375
22.17.3	Menus and the Keyboard .....	376
22.17.4	Menu Example .....	376
22.17.5	The Menu Bar .....	377
22.17.6	Tool bars .....	378
22.17.7	Modifying Menus .....	381
<b>23</b>	<b>Major and Minor Modes .....</b>	<b>382</b>
23.1	Hooks .....	382
23.2	Major Modes .....	384
23.2.1	Major Mode Basics .....	384
23.2.2	Major Mode Conventions .....	385
23.2.3	How Emacs Chooses a Major Mode .....	388
23.2.4	Getting Help about a Major Mode .....	390
23.2.5	Defining Derived Modes.....	391
23.2.6	Generic Modes.....	392
23.2.7	Mode Hooks .....	393
23.2.8	Major Mode Examples .....	394
23.3	Minor Modes .....	397
23.3.1	Conventions for Writing Minor Modes .....	397
23.3.2	Keymaps and Minor Modes .....	399

23.3.3 Defining Minor Modes .....	399
23.4 Mode-Line Format .....	402
23.4.1 Mode Line Basics .....	402
23.4.2 The Data Structure of the Mode Line .....	402
23.4.3 The Top Level of Mode Line Control .....	404
23.4.4 Variables Used in the Mode Line .....	405
23.4.5 %-Constructs in the Mode Line .....	407
23.4.6 Properties in the Mode Line .....	409
23.4.7 Window Header Lines .....	409
23.4.8 Emulating Mode-Line Formatting .....	410
23.5 Imenu .....	410
23.6 Font Lock Mode .....	412
23.6.1 Font Lock Basics .....	413
23.6.2 Search-based Fontification .....	414
23.6.3 Customizing Search-Based Fontification .....	417
23.6.4 Other Font Lock Variables .....	418
23.6.5 Levels of Font Lock .....	419
23.6.6 Precalculated Fontification .....	419
23.6.7 Faces for Font Lock .....	419
23.6.8 Syntactic Font Lock .....	420
23.6.9 Setting Syntax Properties .....	421
23.6.10 Multiline Font Lock Constructs .....	422
23.6.10.1 Font Lock Multiline .....	423
23.6.10.2 Region to Fontify after a Buffer Change .....	423
23.7 Desktop Save Mode .....	424
<b>24 Documentation .....</b>	<b>425</b>
24.1 Documentation Basics .....	425
24.2 Access to Documentation Strings .....	426
24.3 Substituting Key Bindings in Documentation .....	428
24.4 Describing Characters for Help Messages .....	429
24.5 Help Functions .....	431
<b>25 Files .....</b>	<b>434</b>
25.1 Visiting Files .....	434
25.1.1 Functions for Visiting Files .....	434
25.1.2 Subroutines of Visiting .....	436
25.2 Saving Buffers .....	437
25.3 Reading from Files .....	440
25.4 Writing to Files .....	441
25.5 File Locks .....	442
25.6 Information about Files .....	443
25.6.1 Testing Accessibility .....	443
25.6.2 Distinguishing Kinds of Files .....	445
25.6.3 Truenames .....	446
25.6.4 Other Information about Files .....	447
25.6.5 How to Locate Files in Standard Places .....	449
25.7 Changing File Names and Attributes .....	450

25.8	File Names .....	453
25.8.1	File Name Components .....	453
25.8.2	Absolute and Relative File Names.....	455
25.8.3	Directory Names .....	456
25.8.4	Functions that Expand Filenames .....	457
25.8.5	Generating Unique File Names .....	459
25.8.6	File Name Completion .....	461
25.8.7	Standard File Names.....	462
25.9	Contents of Directories.....	462
25.10	Creating and Deleting Directories .....	464
25.11	Making Certain File Names “Magic” .....	464
25.12	File Format Conversion.....	468
<b>26</b>	<b>Backups and Auto-Saving.....</b>	<b>471</b>
26.1	Backup Files.....	471
26.1.1	Making Backup Files.....	471
26.1.2	Backup by Renaming or by Copying?.....	473
26.1.3	Making and Deleting Numbered Backup Files .....	474
26.1.4	Naming Backup Files .....	474
26.2	Auto-Saving .....	476
26.3	Reverting.....	479
<b>27</b>	<b>Buffers .....</b>	<b>481</b>
27.1	Buffer Basics .....	481
27.2	The Current Buffer .....	481
27.3	Buffer Names.....	484
27.4	Buffer File Name .....	485
27.5	Buffer Modification .....	487
27.6	Buffer Modification Time .....	488
27.7	Read-Only Buffers .....	489
27.8	The Buffer List .....	490
27.9	Creating Buffers.....	492
27.10	Killing Buffers.....	493
27.11	Indirect Buffers.....	494
27.12	The Buffer Gap.....	495
<b>28</b>	<b>Windows.....</b>	<b>497</b>
28.1	Basic Concepts of Emacs Windows .....	497
28.2	Splitting Windows .....	498
28.3	Deleting Windows .....	501
28.4	Selecting Windows .....	502
28.5	Cyclic Ordering of Windows .....	503
28.6	Buffers and Windows .....	505
28.7	Displaying Buffers in Windows .....	506
28.8	Choosing a Window for Display .....	508
28.9	Windows and Point .....	511
28.10	The Window Start Position .....	512

28.11	Textual Scrolling .....	515
28.12	Vertical Fractional Scrolling.....	518
28.13	Horizontal Scrolling.....	518
28.14	The Size of a Window .....	520
28.15	Changing the Size of a Window .....	522
28.16	Coordinates and Windows .....	524
28.17	The Window Tree .....	525
28.18	Window Configurations.....	526
28.19	Hooks for Window Scrolling and Changes .....	527
<b>29</b>	<b>Frames.....</b>	<b>529</b>
29.1	Creating Frames.....	529
29.2	Multiple Displays.....	530
29.3	Frame Parameters .....	531
29.3.1	Access to Frame Parameters .....	531
29.3.2	Initial Frame Parameters.....	532
29.3.3	Window Frame Parameters.....	533
29.3.3.1	Basic Parameters.....	533
29.3.3.2	Position Parameters.....	533
29.3.3.3	Size Parameters .....	534
29.3.3.4	Layout Parameters .....	534
29.3.3.5	Buffer Parameters.....	535
29.3.3.6	Window Management Parameters .....	536
29.3.3.7	Cursor Parameters .....	536
29.3.3.8	Color Parameters .....	537
29.3.4	Frame Size And Position .....	538
29.3.5	Geometry .....	540
29.4	Frame Titles.....	540
29.5	Deleting Frames .....	541
29.6	Finding All Frames .....	541
29.7	Frames and Windows .....	542
29.8	Minibuffers and Frames .....	543
29.9	Input Focus .....	543
29.10	Visibility of Frames.....	545
29.11	Raising and Lowering Frames .....	546
29.12	Frame Configurations .....	546
29.13	Mouse Tracking.....	547
29.14	Mouse Position .....	547
29.15	Pop-Up Menus .....	548
29.16	Dialog Boxes .....	549
29.17	Pointer Shape .....	550
29.18	Window System Selections.....	550
29.19	Drag and Drop .....	552
29.20	Color Names .....	552
29.21	Text Terminal Colors .....	554
29.22	X Resources .....	555
29.23	Display Feature Testing.....	555

<b>30 Positions . . . . .</b>	<b>559</b>
30.1 Point . . . . .	559
30.2 Motion . . . . .	560
30.2.1 Motion by Characters . . . . .	560
30.2.2 Motion by Words . . . . .	561
30.2.3 Motion to an End of the Buffer . . . . .	561
30.2.4 Motion by Text Lines . . . . .	562
30.2.5 Motion by Screen Lines . . . . .	564
30.2.6 Moving over Balanced Expressions . . . . .	566
30.2.7 Skipping Characters . . . . .	567
30.3 Excursions . . . . .	568
30.4 Narrowing . . . . .	569
<b>31 Markers . . . . .</b>	<b>572</b>
31.1 Overview of Markers . . . . .	572
31.2 Predicates on Markers . . . . .	573
31.3 Functions that Create Markers . . . . .	573
31.4 Information from Markers . . . . .	575
31.5 Marker Insertion Types . . . . .	576
31.6 Moving Marker Positions . . . . .	576
31.7 The Mark . . . . .	577
31.8 The Region . . . . .	579
<b>32 Text . . . . .</b>	<b>581</b>
32.1 Examining Text Near Point . . . . .	581
32.2 Examining Buffer Contents . . . . .	582
32.3 Comparing Text . . . . .	584
32.4 Inserting Text . . . . .	585
32.5 User-Level Insertion Commands . . . . .	586
32.6 Deleting Text . . . . .	587
32.7 User-Level Deletion Commands . . . . .	589
32.8 The Kill Ring . . . . .	591
32.8.1 Kill Ring Concepts . . . . .	591
32.8.2 Functions for Killing . . . . .	592
32.8.3 Yanking . . . . .	592
32.8.4 Functions for Yanking . . . . .	593
32.8.5 Low-Level Kill Ring . . . . .	594
32.8.6 Internals of the Kill Ring . . . . .	595
32.9 Undo . . . . .	596
32.10 Maintaining Undo Lists . . . . .	598
32.11 Filling . . . . .	599
32.12 Margins for Filling . . . . .	602
32.13 Adaptive Fill Mode . . . . .	603
32.14 Auto Filling . . . . .	604
32.15 Sorting Text . . . . .	605
32.16 Counting Columns . . . . .	609
32.17 Indentation . . . . .	609

32.17.1	Indentation Primitives .....	610
32.17.2	Indentation Controlled by Major Mode.....	610
32.17.3	Indenting an Entire Region .....	611
32.17.4	Indentation Relative to Previous Lines .....	612
32.17.5	Adjustable “Tab Stops” .....	613
32.17.6	Indentation-Based Motion Commands.....	613
32.18	Case Changes.....	613
32.19	Text Properties.....	615
32.19.1	Examining Text Properties.....	615
32.19.2	Changing Text Properties.....	616
32.19.3	Text Property Search Functions.....	618
32.19.4	Properties with Special Meanings .....	620
32.19.5	Formatted Text Properties .....	624
32.19.6	Stickiness of Text Properties .....	625
32.19.7	Saving Text Properties in Files.....	626
32.19.8	Lazy Computation of Text Properties .....	627
32.19.9	Defining Clickable Text .....	628
32.19.10	Links and Mouse-1.....	629
32.19.11	Defining and Using Fields .....	630
32.19.12	Why Text Properties are not Intervals .....	632
32.20	Substituting for a Character Code .....	633
32.21	Registers .....	634
32.22	Transposition of Text .....	635
32.23	Base 64 Encoding.....	635
32.24	MD5 Checksum .....	636
32.25	Atomic Change Groups .....	637
32.26	Change Hooks .....	638
<b>33</b>	<b>Non-ASCII Characters.....</b>	<b>640</b>
33.1	Text Representations .....	640
33.2	Converting Text Representations.....	641
33.3	Selecting a Representation .....	642
33.4	Character Codes.....	643
33.5	Character Sets.....	644
33.6	Characters and Bytes .....	645
33.7	Splitting Characters .....	645
33.8	Scanning for Character Sets.....	646
33.9	Translation of Characters .....	647
33.10	Coding Systems .....	648
33.10.1	Basic Concepts of Coding Systems .....	648
33.10.2	Encoding and I/O .....	649
33.10.3	Coding Systems in Lisp .....	650
33.10.4	User-Chosen Coding Systems .....	652
33.10.5	Default Coding Systems.....	653
33.10.6	Specifying a Coding System for One Operation .....	655
33.10.7	Explicit Encoding and Decoding .....	656
33.10.8	Terminal I/O Encoding .....	657
33.10.9	MS-DOS File Types .....	658

33.11 Input Methods .....	659
33.12 Locales .....	660
<b>34 Searching and Matching.....</b>	<b>661</b>
34.1 Searching for Strings.....	661
34.2 Searching and Case .....	663
34.3 Regular Expressions .....	663
34.3.1 Syntax of Regular Expressions .....	663
34.3.1.1 Special Characters in Regular Expressions .....	664
34.3.1.2 Character Classes .....	667
34.3.1.3 Backslash Constructs in Regular Expressions .....	668
34.3.2 Complex Regexp Example .....	671
34.3.3 Regular Expression Functions .....	672
34.4 Regular Expression Searching .....	673
34.5 POSIX Regular Expression Searching .....	676
34.6 The Match Data .....	676
34.6.1 Replacing the Text that Matched.....	676
34.6.2 Simple Match Data Access .....	677
34.6.3 Accessing the Entire Match Data.....	679
34.6.4 Saving and Restoring the Match Data.....	680
34.7 Search and Replace .....	681
34.8 Standard Regular Expressions Used in Editing .....	683
<b>35 Syntax Tables .....</b>	<b>684</b>
35.1 Syntax Table Concepts.....	684
35.2 Syntax Descriptors.....	684
35.2.1 Table of Syntax Classes .....	685
35.2.2 Syntax Flags.....	687
35.3 Syntax Table Functions .....	688
35.4 Syntax Properties.....	690
35.5 Motion and Syntax .....	691
35.6 Parsing Expressions.....	691
35.6.1 Motion Commands Based on Parsing.....	692
35.6.2 Finding the Parse State for a Position .....	692
35.6.3 Parser State .....	693
35.6.4 Low-Level Parsing .....	694
35.6.5 Parameters to Control Parsing .....	695
35.7 Some Standard Syntax Tables.....	695
35.8 Syntax Table Internals .....	695
35.9 Categories .....	696
<b>36 Abbrevs and Abbrev Expansion .....</b>	<b>699</b>
36.1 Setting Up Abbrev Mode .....	699
36.2 Abbrev Tables .....	699
36.3 Defining Abbrevs .....	700
36.4 Saving Abbrevs in Files .....	701
36.5 Looking Up and Expanding Abbreviations .....	702
36.6 Standard Abbrev Tables .....	704

<b>37 Processes . . . . .</b>	<b>705</b>
37.1 Functions that Create Subprocesses . . . . .	705
37.2 Shell Arguments . . . . .	706
37.3 Creating a Synchronous Process . . . . .	707
37.4 Creating an Asynchronous Process . . . . .	710
37.5 Deleting Processes . . . . .	712
37.6 Process Information . . . . .	712
37.7 Sending Input to Processes . . . . .	714
37.8 Sending Signals to Processes . . . . .	715
37.9 Receiving Output from Processes . . . . .	717
37.9.1 Process Buffers . . . . .	717
37.9.2 Process Filter Functions . . . . .	718
37.9.3 Decoding Process Output . . . . .	720
37.9.4 Accepting Output from Processes . . . . .	721
37.10 Sentinels: Detecting Process Status Changes . . . . .	721
37.11 Querying Before Exit . . . . .	723
37.12 Transaction Queues . . . . .	723
37.13 Network Connections . . . . .	724
37.14 Network Servers . . . . .	726
37.15 Datagrams . . . . .	726
37.16 Low-Level Network Access . . . . .	727
37.16.1 <code>make-network-process</code> . . . . .	727
37.16.2 Network Options . . . . .	729
37.16.3 Testing Availability of Network Features . . . . .	730
37.17 Misc Network Facilities . . . . .	731
37.18 Packing and Unpacking Byte Arrays . . . . .	732
37.18.1 Describing Data Layout . . . . .	732
37.18.2 Functions to Unpack and Pack Bytes . . . . .	734
37.18.3 Examples of Byte Unpacking and Packing . . . . .	735
<b>38 Emacs Display . . . . .</b>	<b>739</b>
38.1 Refreshing the Screen . . . . .	739
38.2 Forcing Redisplay . . . . .	739
38.3 Truncation . . . . .	740
38.4 The Echo Area . . . . .	741
38.4.1 Displaying Messages in the Echo Area . . . . .	741
38.4.2 Reporting Operation Progress . . . . .	743
38.4.3 Logging Messages in ‘*Messages*’ . . . . .	744
38.4.4 Echo Area Customization . . . . .	745
38.5 Reporting Warnings . . . . .	745
38.5.1 Warning Basics . . . . .	746
38.5.2 Warning Variables . . . . .	746
38.5.3 Warning Options . . . . .	748
38.6 Invisible Text . . . . .	748
38.7 Selective Display . . . . .	750
38.8 Temporary Displays . . . . .	752
38.9 Overlays . . . . .	754
38.9.1 Managing Overlays . . . . .	754

38.9.2	Overlay Properties .....	756
38.9.3	Searching for Overlays .....	759
38.10	Width .....	760
38.11	Line Height .....	761
38.12	Faces .....	762
38.12.1	Defining Faces .....	763
38.12.2	Face Attributes .....	765
38.12.3	Face Attribute Functions .....	767
38.12.4	Displaying Faces .....	770
38.12.5	Font Selection .....	771
38.12.6	Functions for Working with Faces .....	772
38.12.7	Automatic Face Assignment .....	773
38.12.8	Looking Up Fonts .....	773
38.12.9	Fontsets .....	774
38.13	Fringes .....	776
38.13.1	Fringe Size and Position .....	776
38.13.2	Fringe Indicators .....	777
38.13.3	Fringe Cursors .....	779
38.13.4	Fringe Bitmaps .....	779
38.13.5	Customizing Fringe Bitmaps .....	780
38.13.6	The Overlay Arrow .....	780
38.14	Scroll Bars .....	781
38.15	The <i>display</i> Property .....	783
38.15.1	Specified Spaces .....	783
38.15.2	Pixel Specification for Spaces .....	784
38.15.3	Other Display Specifications .....	785
38.15.4	Displaying in the Margins .....	786
38.16	Images .....	787
38.16.1	Image Descriptors .....	788
38.16.2	XBM Images .....	791
38.16.3	XPM Images .....	792
38.16.4	GIF Images .....	792
38.16.5	PostScript Images .....	792
38.16.6	Other Image Types .....	792
38.16.7	Defining Images .....	793
38.16.8	Showing Images .....	795
38.16.9	Image Cache .....	796
38.17	Buttons .....	796
38.17.1	Button Properties .....	797
38.17.2	Button Types .....	798
38.17.3	Making Buttons .....	798
38.17.4	Manipulating Buttons .....	799
38.17.5	Button Buffer Commands .....	800
38.18	Abstract Display .....	800
38.18.1	Abstract Display Functions .....	801
38.18.2	Abstract Display Example .....	803
38.19	Blinking Parentheses .....	805
38.20	Usual Display Conventions .....	806

38.21	Display Tables .....	807
38.21.1	Display Table Format .....	807
38.21.2	Active Display Table.....	809
38.21.3	Glyphs .....	809
38.22	Beeping .....	810
38.23	Window Systems .....	811
<b>39</b>	<b>Operating System Interface .....</b>	<b>812</b>
39.1	Starting Up Emacs .....	812
39.1.1	Summary: Sequence of Actions at Startup.....	812
39.1.2	The Init File, ‘.emacs’ .....	813
39.1.3	Terminal-Specific Initialization .....	814
39.1.4	Command-Line Arguments.....	815
39.2	Getting Out of Emacs .....	817
39.2.1	Killing Emacs.....	817
39.2.2	Suspending Emacs .....	817
39.3	Operating System Environment .....	819
39.4	User Identification .....	822
39.5	Time of Day .....	824
39.6	Time Conversion.....	825
39.7	Parsing and Formatting Times .....	826
39.8	Processor Run time.....	828
39.9	Time Calculations .....	828
39.10	Timers for Delayed Execution.....	829
39.11	Idle Timers .....	831
39.12	Terminal Input .....	832
39.12.1	Input Modes.....	832
39.12.2	Recording Input .....	833
39.13	Terminal Output .....	834
39.14	Sound Output .....	835
39.15	Operating on X11 Keysyms .....	835
39.16	Batch Mode .....	836
39.17	Session Management.....	836
<b>Appendix A</b>	<b>Emacs 21 Antinews .....</b>	<b>838</b>
A.1	Old Lisp Features in Emacs 21 .....	838
<b>Appendix B</b>	<b>GNU Free Documentation License .....</b>	<b>843</b>
	ADDENDUM: How to use this License for your documents .....	849
<b>Appendix C</b>	<b>GNU General Public License... 850</b>	
	Preamble .....	850
	Terms and Conditions for Copying, Distribution and Modification ..	851
	How to Apply These Terms to Your New Programs .....	855

<b>Appendix D Tips and Conventions.....</b>	<b>856</b>
D.1 Emacs Lisp Coding Conventions.....	856
D.2 Key Binding Conventions.....	859
D.3 Emacs Programming Tips.....	860
D.4 Tips for Making Compiled Code Fast.....	861
D.5 Tips for Avoiding Compiler Warnings.....	862
D.6 Tips for Documentation Strings.....	862
D.7 Tips on Writing Comments.....	865
D.8 Conventional Headers for Emacs Libraries.....	867
<b>Appendix E GNU Emacs Internals .....</b>	<b>870</b>
E.1 Building Emacs .....	870
E.2 Pure Storage .....	871
E.3 Garbage Collection .....	872
E.4 Memory Usage .....	875
E.5 Writing Emacs Primitives.....	876
E.6 Object Internals .....	880
E.6.1 Buffer Internals.....	880
E.6.2 Window Internals.....	886
E.6.3 Process Internals .....	889
<b>Appendix F Standard Errors .....</b>	<b>891</b>
<b>Appendix G Buffer-Local Variables .....</b>	<b>895</b>
<b>Appendix H Standard Keymaps.....</b>	<b>899</b>
<b>Appendix I Standard Hooks .....</b>	<b>903</b>
<b>Index.....</b>	<b>908</b>

# 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 edition 2.9 of the GNU Emacs Lisp Reference Manual, corresponding to Emacs version 22.1.

## 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. This includes most information about usage on VMS.

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 number of the edition you are criticizing.

Please mail comments and corrections to

`bug-lisp-manual@gnu.org`

We let mail to this list accumulate unread until someone decides to apply the corrections. Months, and sometimes years, go by between updates. So please attach no significance to the lack of a reply—your mail *will* be acted on in due time. If you want to contact the Emacs maintainers more quickly, send mail to `bug-gnu-emacs@gnu.org`.

## 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 implementors 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 ‘c1’ library. See Info file ‘c1’, node ‘Top’.

Emacs Lisp is not at all influenced by Scheme; but the GNU project has an implementation of Scheme, called Guile. We use Guile 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 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 which 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 11.2 [Constant Variables], page 135.

**booleanp object** [Function]  
Return non-`nil` iff `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
```

Sometimes to help describe one form we 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*`’), 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 (here `bar`) 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` will receive, 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...)
```

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. Arguments named `object` may be of any type. (See Chapter 2 [Lisp Data Types], page 8, for a list of Emacs object types.) Arguments with other sorts of names (e.g., `new-file`) are discussed specifically in the description of the function. In some sections, features common to the arguments of several functions are described at the beginning.

See Section 12.2 [Lambda Expressions], page 161, for a more complete description of optional and rest arguments.

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.

Special form descriptions 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 hold a 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 20.3.5 (i486-pc-linux-gnulibc1, X toolkit)
      of Sat Feb 14 1998 on psilocin.gnu.org"
```

If *here* is non-*nil*, it inserts the text in the buffer before point, and returns *nil*. Called interactively, the function 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 at the local site. It is a list of three integers, like the value of `current-time` (see Section 39.5 [Time of Day], page 824).

```
emacs-build-time
  ⇒ (13623 62065 344633)
```

`emacs-version`

[Variable]

The value of this variable is the version of Emacs being run. It is a string such as “20.3.1”. The last number in this string is not really part of the Emacs release version number; it is incremented each time you build Emacs in any given directory. A value with four numeric components, such as “20.3.9.1”, indicates an unreleased test version.

The following two variables have existed since Emacs version 19.23:

**emacs-major-version** [Variable]

The major version number of Emacs, as an integer. For Emacs version 20.3, the value is 20.

**emacs-minor-version** [Variable]

The minor version number of Emacs, as an integer. For Emacs version 20.3, the value is 3.

## 1.5 Acknowledgements

This manual was 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.

Corrections were supplied by 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, 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, Skip Montanaro, John Gardiner Myers, Thomas A. Peterson, Francesco Potorti, Friedrich Pukelsheim, Arnold D. Robbins, Raul Rockwell, Per Starbäck, Shinichirou Sugou, Kimmo Suominen, Edward Tharp, Bill Trost, Rickard Westman, Jean White, Matthew Wilding, Carl Witty, Dale Worley, Rusty Wright, and David D. Zuhn.

## 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*, and *byte-code function*, plus several special types, such as *buffer*, that are related to editing. (See Section 2.4 [Editing Types], page 23.)

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

Note that Lisp is unlike many other languages in that Lisp objects are *self-typing*: the primitive type of the 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 11.15 [Variables with Restricted Values], page 158.)

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 19 [Read and Print], page 268.

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 ‘>’. 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 ‘#<’.

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 9 [Evaluation], page 110). 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 19.3 [Input Functions], page 271, for a description of `read`, the basic function for reading objects.

## 2.2 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, a 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 16 [Byte Compilation], page 214). It isn't meant for source files, however.

See Section D.7 [Comment Tips], page 865, for conventions for formatting comments.

## 2.3 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.

### 2.3.1 Integer Type

The range of values for integers in Emacs Lisp is  $-2^{28}$  to  $2^{28} - 1$  (29 bits; i.e.,  $-2^{28}$  to  $2^{28} - 1$ ) on most machines. (Some machines may provide a wider range.) It is important to note that the Emacs Lisp arithmetic functions do not check for overflow. Thus (`(1+ 268435455)`) is  $-2^{28}$  on most machines.

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 ‘.’.

<code>-1</code>	; The integer -1.
<code>1</code>	; The integer 1.
<code>1.</code>	; Also the integer 1.
<code>+1</code>	; Also the integer 1.
<code>536870913</code>	; Also the integer 1 on a 29-bit implementation.

See Chapter 3 [Numbers], page 32, for more information.

### 2.3.2 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’, ‘15.0e2’, ‘1.5e3’, and ‘.15e4’ are five ways of writing a floating point number whose value is 1500. They are all equivalent.

See Chapter 3 [Numbers], page 32, for more information.

### 2.3.3 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.

Individual characters are used occasionally in programs, but it is more common to work with *strings*, which are sequences composed of characters. See Section 2.3.8 [String Type], page 18.

Characters in strings, buffers, and files are currently limited to the range of 0 to 524287—nineteen bits. But not all values in that range are valid character codes. Codes 0 through 127 are ASCII codes; the rest are non-ASCII (see Chapter 33 [Non-ASCII Characters], page 640). 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 24.4 [Describing Characters], page 429.

#### 2.3.3.1 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, but it is often a good idea to add a ‘\’ so that the Emacs commands for editing Lisp code don’t get confused. For example, ‘?\(’ is the way to write the open-paren character. 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, <i>C-g</i>
?\\b	⇒ 8	; backspace, BS, <i>C-h</i>
?\\t	⇒ 9	; tab, TAB, <i>C-i</i>
?\\n	⇒ 10	; newline, <i>C-j</i>
?\\v	⇒ 11	; vertical tab, <i>C-k</i>
?\\f	⇒ 12	; formfeed character, <i>C-l</i>
?\\r	⇒ 13	; carriage return, RET, <i>C-m</i>
?\\e	⇒ 27	; escape character, ESC, <i>C-[</i>
?\\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 terminology 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 should add a backslash before any of the characters ‘()\\|;’ ‘"#.,’ to avoid confusing the Emacs commands for editing Lisp code. 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.)

### 2.3.3.2 General Escape Syntax

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

For instance, you can specify characters by their Unicode values. ‘?\\unnnnn’ represents a character that maps to the Unicode code point ‘U+nnnn’. There is a slightly different syntax for specifying characters with code points above #xFFFF; ‘\\U00nnnnnn’ represents the character whose Unicode code point is ‘U+nnnnnn’, if such a character is supported by Emacs. If the corresponding character is not supported, Emacs signals an error.

This peculiar and inconvenient syntax was adopted for compatibility with other programming languages. Unlike some other languages, Emacs Lisp supports this syntax in only character literals and strings.

The most general read syntax for a character represents the character code in either octal or hex. To use octal, write a question mark followed by a backslash and the octal character code (up to three octal digits); thus, ‘?\\101’ for the character *A*, ‘?\\001’ for the character *C-a*, and ‘?\\002’ for the character *C-b*. Although this syntax can represent any ASCII character, it is preferred only when the precise octal value is more important than the ASCII representation.

<code>?\\012</code> ⇒ 10	<code>?\\n</code> ⇒ 10	<code>?\\C-j</code> ⇒ 10
<code>?\\101</code> ⇒ 65	<code>?A</code> ⇒ 65	

To use hex, write a question mark followed by a backslash, ‘x’, and the hexadecimal character code. You can use any number of hex digits, so you can represent any character code in this way. Thus, ‘?\\x41’ for the character *A*, ‘?\\x1’ for the character *C-a*, and ?\\x8e0 for the Latin-1 character ‘à’.

### 2.3.3.3 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. Ordinary terminals have no way of generating non-ASCII control characters, but you can generate them straightforwardly 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.

### 2.3.3.4 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. (In Emacs versions 18 and older, this convention was used for characters outside of strings as well.)

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’.

### 2.3.3.5 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 when you use X terminals or other special terminals; ordinary terminals do not report the distinction to the computer in any way. The Lisp syntax for the shift bit is ‘\S-’; thus, ‘?\C-\S-o’ or ‘?\C-\S-0’ 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.

### 2.3.4 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 8.3 [Creating Symbols], page 104, 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.

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 fifth example is escaped to prevent it from being read as a number. This is not necessary in the fourth 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’.
char-to-string ; A symbol named ‘char-to-string’.
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.

```

Normally the Lisp reader interns all symbols (see Section 8.3 [Creating Symbols], page 104). To prevent interning, you can write ‘#:’ before the name of the symbol.

### 2.3.5 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. Thus, an object of type list or of type array is also considered a sequence.

Arrays are further subdivided into strings, vectors, char-tables and bool-vectors. Vectors can hold elements of any type, but 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 32.19 [Text Properties], page 615), but vectors do not support text properties, even when their elements happen to be characters.

Lists, strings and the other array types are different, but they have 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 `elt` can be used to extract an element of a sequence, given its index. See Chapter 6 [Sequences Arrays Vectors], page 87.

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`.

### 2.3.6 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* or *refer to* 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 note to C programmers: in Lisp, we do not distinguish between “holding” a value and “pointing to” the value, because pointers in Lisp are implicit.

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 63, for functions that work on lists. Because most cons cells are used as part of lists, the phrase *list structure* has come to refer to any structure made out of cons cells.

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:

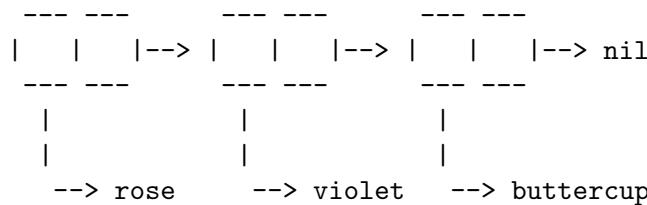
(A 2 "A")	; A list of three elements.
()	; A list of no elements (the empty list).
nil	; A list of no elements (the empty list).
("A ()")	; A list of one element: the string "A ()".
(A ())	; A list of two elements: A and the empty list.
(A nil)	; Equivalent to the previous.
((A B C))	; A list of one 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, called the “address” part 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.

### 2.3.6.1 Drawing Lists as Box Diagrams

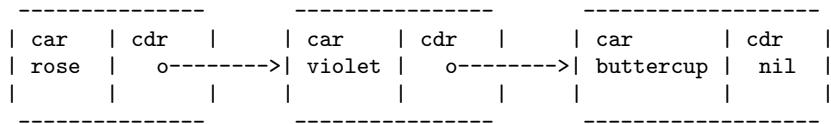
A list can be illustrated by a diagram in which the cons cells are shown as pairs of boxes, like dominoes. (The Lisp reader cannot read such an illustration; unlike the textual notation, which can be understood by both humans and computers, the box illustrations can be understood only by humans.) This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

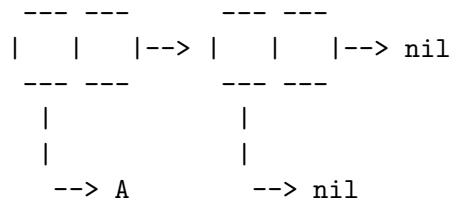
In this example, the first box, which holds the CAR of the first cons cell, refers to or “holds” `rose` (a symbol). The second box, holding the CDR of the first cons cell, refers to the next pair of boxes, the second cons cell. The CAR of the second cons cell is `violet`, and its CDR is the third cons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

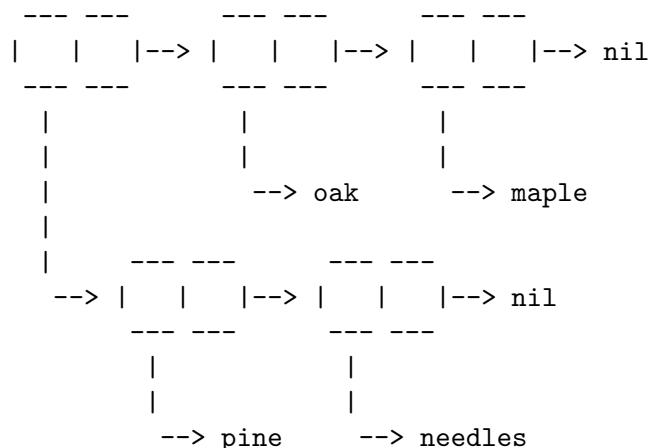


A list with no elements in it is the *empty list*; it is identical to the symbol `nil`. In other words, `nil` is both a symbol and a list.

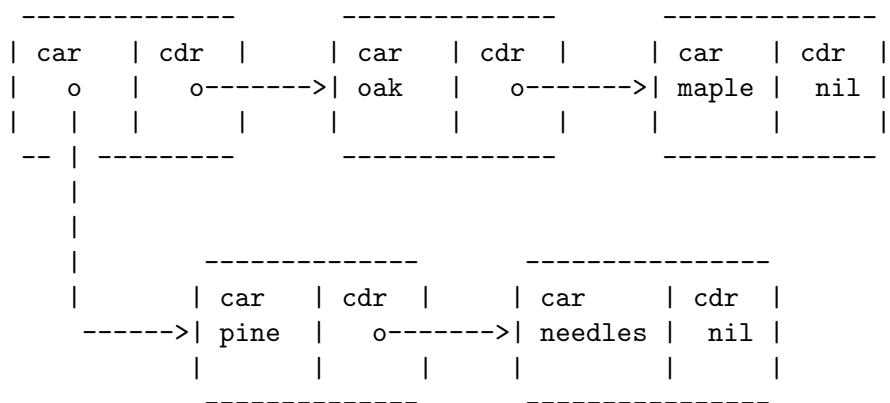
Here is the list (A ()), or equivalently (A nil), depicted with boxes and arrows:



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:

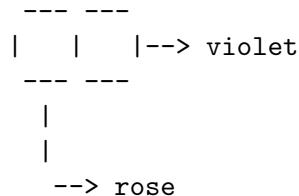


### 2.3.6.2 Dotted Pair Notation

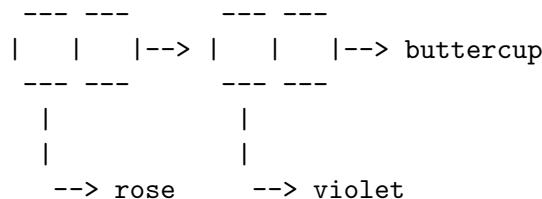
*Dotted pair notation* is a general syntax for cons cells that represents the CAR and CDR explicitly. In this syntax,  $(a . b)$  stands for a cons cell whose CAR is the object  $a$  and

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)`:

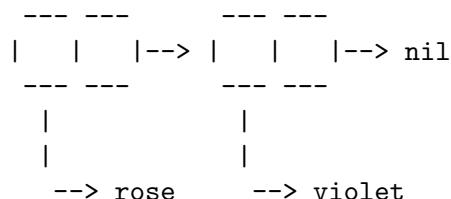


You can combine dotted pair notation with list notation to represent conveniently a chain of cons cells with a non-`nil` final CDR. You write a dot after the last element of the list, followed by the CDR of the final cons cell. For example, `(rose violet . buttercup)` is equivalent to `(rose . (violet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

### 2.3.6.3 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 81, for a further explanation of alists and for functions that work on alists. See Chapter 7 [Hash Tables], page 97, for another kind of lookup table, which is much faster for handling a large number of keys.

### 2.3.7 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 integer. 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.

### 2.3.8 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 47, for functions that operate on strings.

#### 2.3.8.1 Syntax for Strings

The read syntax for strings 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 a single double-quote character. Likewise, you can include a backslash by preceding it with another backslash, like this: `"this \\ is a single embedded backslash"`.

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."
```

### 2.3.8.2 Non-ASCII Characters in Strings

You can include a non-ASCII international character in a string constant by writing it literally. There are two text representations for non-ASCII characters in Emacs strings (and in buffers): unibyte and multibyte. 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 the character is read as a multibyte character, and that makes the string multibyte. If the string constant is read from a unibyte source, then the character is read as unibyte and that makes the string unibyte.

You can also represent a multibyte non-ASCII character with its character code: use a hex escape, ‘\xnnnnnnn’, with as many digits as necessary. (Multibyte non-ASCII character codes are all greater than 256.) Any character which is not a valid hex digit terminates this construct. If the next character in the string could be interpreted as a hex digit, write ‘\ ’ (backslash and space) to terminate the hex escape—for example, ‘\x8e0\ ’ 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 the preceding hex escape.

You can represent a unibyte non-ASCII character with its character code, which must be in the range from 128 (0200 octal) to 255 (0377 octal). If you write all such character codes in octal and the string contains no other characters forcing it to be multibyte, this produces a unibyte string. However, using any hex escape in a string (even for an ASCII character) forces the string to be multibyte.

You can also specify characters in a string by their numeric values in Unicode, using ‘\u’ and ‘\U’ (see Section 2.3.3 [Character Type], page 10).

See Section 33.1 [Text Representations], page 640, for more information about the two text representations.

### 2.3.8.3 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 2.3.3 [Character Type], page 10, for a description of the 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<sup>7</sup> 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 2.3.3 [Character Type], page 10.

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

#### 2.3.8.4 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 32.19 [Text Properties], page 615, 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.)

#### 2.3.9 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 91, for functions that work with vectors.

#### 2.3.10 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.

See Section 6.6 [Char-Tables], page 93, for special functions to operate on char-tables. Uses of char-tables include:

- Case tables (see Section 4.9 [Case Tables], page 60).
- Character category tables (see Section 35.9 [Categories], page 696).
- Display tables (see Section 38.21 [Display Tables], page 807).
- Syntax tables (see Chapter 35 [Syntax Tables], page 684).

### 2.3.11 Bool-Vector Type

A bool-vector is a one-dimensional array of elements that 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

```

### 2.3.12 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. Hash tables have no read syntax, and print using hash notation. See Chapter 7 [Hash Tables], page 97, for functions that operate on hash tables.

```
(make-hash-table)
  ⇒ #<hash-table 'eql nil 0/65 0x83af980>

```

### 2.3.13 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 12.2 [Lambda Expressions], page 161).

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 12.7 [Anonymous Functions], page 170). A named function in Lisp is just a symbol with a valid function in its function cell (see Section 12.4 [Defining Functions], page 165).

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 12.5 [Calling Functions], page 167.

### 2.3.14 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` function, but any list that begins with `macro` is a macro as far as Emacs is concerned. See Chapter 13 [Macros], page 176, for an explanation of how to write a macro.

**Warning:** Lisp macros and keyboard macros (see Section 21.15 [Keyboard Macros], page 345) are entirely different things. When we use the word “macro” without qualification, we mean a Lisp macro, not a keyboard macro.

### 2.3.15 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 9.1.7 [Special Forms], page 114).

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 2.3.13 [Function Type], page 21, 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.
```

### 2.3.16 Byte-Code Function Type

The byte compiler produces *byte-code function objects*. Internally, a byte-code function object is much like a vector; however, the evaluator handles this data type specially when

it appears as a function to be called. See Chapter 16 [Byte Compilation], page 214, for information about the byte compiler.

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

### 2.3.17 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 15.5 [Autoload], page 206, for more details.

## 2.4 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.

### 2.4.1 Buffer Type

A *buffer* is an object that holds text that can be edited (see Chapter 27 [Buffers], page 481). Most buffers hold the contents of a disk file (see Chapter 25 [Files], page 434) 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 28 [Windows], page 497). But a buffer need not be displayed in any window.

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.

Each buffer has a designated position called *point* (see Chapter 30 [Positions], page 559). At any time, one buffer is the *current buffer*. Most editing commands act on the contents of the current buffer in the neighborhood of point. 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 32 [Text], page 581).

Several other data structures are associated with each buffer:

- a local syntax table (see Chapter 35 [Syntax Tables], page 684);
- a local keymap (see Chapter 22 [Keymaps], page 347); and,
- a list of buffer-local variable bindings (see Section 11.10 [Buffer-Local Variables], page 147).

- overlays (see Section 38.9 [Overlays], page 754).
- text properties for the text in the buffer (see Section 32.19 [Text Properties], page 615).

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 27.11 [Indirect Buffers], page 494.

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

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

### 2.4.2 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 31 [Markers], page 572, for information on how to test, create, copy, and move markers.

### 2.4.3 Window Type

A *window* describes the portion of the terminal screen that Emacs uses to display a buffer. Every window has one associated buffer, whose contents appear in the window. By contrast, a given buffer may appear in one window, no window, or several windows.

Though many windows may exist simultaneously, at any time one window is designated the *selected window*. 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, but this is not necessarily the case.

Windows are grouped on the screen into frames; each window belongs to one and only one frame. See Section 2.4.4 [Frame Type], page 25.

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 28 [Windows], page 497, for a description of the functions that work on windows.

#### 2.4.4 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 29 [Frames], page 529, for a description of the functions that work on frames.

#### 2.4.5 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 28.18 [Window Configurations], page 526, for a description of several functions related to window configurations.

#### 2.4.6 Frame Configuration Type

A *frame configuration* stores information about the positions, sizes, and contents of the windows in all frames. 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 29.12 [Frame Configurations], page 546, for a description of several functions related to frame configurations.

#### 2.4.7 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 37 [Processes], page 705, for information about functions that create, delete, return information about, send input or signals to, and receive output from processes.

#### 2.4.8 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 20 [Minibuffers], page 278) or output in the echo area (see Section 38.4 [The Echo Area], page 741).

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

See Chapter 19 [Read and Print], page 268, for a description of functions related to streams, including parsing and printing functions.

### 2.4.9 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 22 [Keymaps], page 347, for information about creating keymaps, handling prefix keys, local as well as global keymaps, and changing key bindings.

### 2.4.10 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 38.9 [Overlays], page 754, for how to create and use overlays.

## 2.5 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 to 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 19.6 [Output Variables], page 276.

## 2.6 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 64.

- arrayp** See Section 6.3 [Array Functions], page 90.
- bool-vector-p**
  - See Section 6.7 [Bool-Vectors], page 95.
- bufferp** See Section 27.1 [Buffer Basics], page 481.
- byte-code-function-p**
  - See Section 2.3.16 [Byte-Code Type], page 22.
- case-table-p**
  - See Section 4.9 [Case Tables], page 60.
- char-or-string-p**
  - See Section 4.2 [Predicates for Strings], page 48.
- char-table-p**
  - See Section 6.6 [Char-Tables], page 93.
- commandp** See Section 21.3 [Interactive Call], page 310.
- consp** See Section 5.2 [List-related Predicates], page 64.
- display-table-p**
  - See Section 38.21 [Display Tables], page 807.
- floatp** See Section 3.3 [Predicates on Numbers], page 34.
- frame-configuration-p**
  - See Section 29.12 [Frame Configurations], page 546.
- frame-live-p**
  - See Section 29.5 [Deleting Frames], page 541.
- framep** See Chapter 29 [Frames], page 529.
- functionp**
  - See Chapter 12 [Functions], page 160.
- hash-table-p**
  - See Section 7.4 [Other Hash], page 100.
- integer-or-marker-p**
  - See Section 31.2 [Predicates on Markers], page 573.
- integerp** See Section 3.3 [Predicates on Numbers], page 34.
- keymapp** See Section 22.4 [Creating Keymaps], page 350.
- keywordp** See Section 11.2 [Constant Variables], page 135.
- listp** See Section 5.2 [List-related Predicates], page 64.
- markerp** See Section 31.2 [Predicates on Markers], page 573.
- wholenump**
  - See Section 3.3 [Predicates on Numbers], page 34.
- nlistp** See Section 5.2 [List-related Predicates], page 64.

**numberp** See Section 3.3 [Predicates on Numbers], page 34.

**number-or-marker-p**  
See Section 31.2 [Predicates on Markers], page 573.

**overlayp** See Section 38.9 [Overlays], page 754.

**processp** See Chapter 37 [Processes], page 705.

**sequencep**  
See Section 6.1 [Sequence Functions], page 87.

**stringp** See Section 4.2 [Predicates for Strings], page 48.

**subrp** See Section 12.8 [Function Cells], page 171.

**symbolp** See Chapter 8 [Symbols], page 102.

**syntax-table-p**  
See Chapter 35 [Syntax Tables], page 684.

**user-variable-p**  
See Section 11.5 [Defining Variables], page 139.

**vectorp** See Section 6.4 [Vectors], page 91.

**window-configuration-p**  
See Section 28.18 [Window Configurations], page 526.

**window-live-p**  
See Section 28.3 [Deleting Windows], page 501.

**windowp** See Section 28.1 [Basic Windows], page 497.

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

**string-or-null-p**  
See Section 4.2 [Predicates for Strings], page 48.

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 `symbol`, `integer`, `float`, `string`, `cons`, `vector`, `char-table`, `bool-vector`, `hash-table`, `subr`, `compiled-function`, `marker`, `overlay`, `window`, `buffer`, `frame`, `process`, or `window-configuration`.

```
(type-of 1)
      ⇒ integer
(type-of 'nil)
      ⇒ symbol
(type-of '())
      ; () is nil.
      ⇒ symbol
(type-of '(x))
      ⇒ cons
```

## 2.7 Equality Predicates

Here we describe two functions that test for equality between any two objects. Other functions test equality 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, `nil` otherwise.

`eq` returns `t` if `object1` and `object2` are integers with the same value. Also, since symbol names are normally unique, if the arguments are symbols with the same name, they are `eq`. For other 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.

```
(eq 'foo 'foo)
⇒ t

(eq 456 456)
⇒ t

(eq "asdf" "asdf")
⇒ nil

(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 8.3 [Creating Symbols], page 104.

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

**equal object1 object2** [Function]

This function returns `t` if `object1` and `object2` have equal components, `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. 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 either in the range 0 through 127 (ASCII) or 160 through 255 (`eight-bit-graphic`). (see Section 33.1 [Text Representations], page 640).

```
(equal "asdf" "ASDF")
⇒ nil
```

However, two distinct buffers are never considered `equal`, even if their textual contents are the same.

The test for equality is implemented 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))
```

Because of this recursive method, circular lists may therefore cause infinite recursion (leading to an error).

## 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  $511$ . Their values are exact. Floating point numbers are numbers with fractional parts, such as  $-4.5$ ,  $0.0$ , or  $2.71828$ . They can also be expressed in exponential notation:  $1.5\text{e}2$  equals  $150$ ; in this example, ‘ $\text{e}2$ ’ stands for ten to the second power, and that is multiplied by  $1.5$ . Floating point values are not exact; they have a fixed, limited amount of precision.

### 3.1 Integer Basics

The range of values for an integer depends on the machine. The minimum range is  $-268435456$  to  $268435455$  (29 bits; i.e.,  $-2^{28}$  to  $2^{28} - 1$ ), but some machines may provide a wider range. Many examples in this chapter assume an integer has 29 bits.

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

<code>1</code>	;	The integer 1.
<code>1.</code>	;	The integer 1.
<code>+1</code>	;	Also the integer 1.
<code>-1</code>	;	The integer $-1$ .
<code>536870913</code>	;	Also the integer 1, due to overflow.
<code>0</code>	;	The integer 0.
<code>-0</code>	;	The integer 0.

The syntax for integers in bases other than 10 uses ‘#’ followed by a letter that specifies the radix: ‘`b`’ for binary, ‘`o`’ for octal, ‘`x`’ for hex, or ‘`radixr`’ to specify radix `radix`. Case is not significant for the letter that specifies the radix. Thus, ‘`#binteger`’ reads `integer` in binary, and ‘`#radixrinteger`’ reads `integer` in radix `radix`. Allowed values of `radix` run from 2 to 36. 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 41), it is often helpful to view the numbers in their binary form.

In 29-bit binary, the decimal integer 5 looks like this:

```
0 0000 0000 0000 0000 0000 0101
```

(We have inserted spaces between groups of 4 bits, and two spaces between groups of 8 bits, to make the binary integer easier to read.)

The integer  $-1$  looks like this:

```
1 1111 1111 1111 1111 1111 1111
```

$-1$  is represented as 29 ones. (This is called *two’s complement* notation.)

The negative integer,  $-5$ , is created by subtracting 4 from  $-1$ . In binary, the decimal integer 4 is 100. Consequently,  $-5$  looks like this:

```
1 1111 1111 1111 1111 1111 1111 1011
```

In this implementation, the largest 29-bit binary integer value is 268,435,455 in decimal. In binary, it looks like this:

```
0 1111 1111 1111 1111 1111 1111 1111
```

Since the arithmetic functions do not check whether integers go outside their range, when you add 1 to 268,435,455, the value is the negative integer -268,435,456:

```
(+ 1 268435455)
⇒ -268435456
⇒ 1 0000 0000 0000 0000 0000 0000 0000
```

Many of the functions described in this chapter accept markers for arguments in place of numbers. (See Chapter 31 [Markers], page 572.) 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.

#### `most-positive-fixnum`

[Variable]

The value of this variable is the largest integer that Emacs Lisp can handle.

#### `most-negative-fixnum`

[Variable]

The value of this variable is the smallest integer that Emacs Lisp can handle. It is negative.

## 3.2 Floating Point Basics

Floating point numbers are useful for representing numbers that are not integral. The precise range of floating point numbers is machine-specific; it is the same as the range of the C data type `double` on the machine you are using.

The read-syntax for floating point numbers requires either a decimal point (with at least one digit following), an exponent, or both. For example, ‘1500.0’, ‘15e2’, ‘15.0e2’, ‘1.5e3’, and ‘.15e4’ are five ways of writing a floating point number whose value is 1500. They are all equivalent. You can also use a minus sign to write negative floating point numbers, as in ‘-1.0’.

Most modern computers support the IEEE floating point standard, which provides for 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. For practical purposes, there’s no significant difference between different NaN values in Emacs Lisp, and there’s no rule for precisely which NaN value should be used in a particular case, so Emacs Lisp doesn’t try to distinguish them (but it does report the sign, if you print it). Here are the read syntaxes for these special floating point values:

positive infinity

‘1.0e+INF’

negative infinity

‘-1.0e+INF’

Not-a-number

‘0.0e+NaN’ or ‘-0.0e+NaN’.

To test whether a floating point value is a NaN, compare it with itself using `=`. That returns `nil` for a NaN, and `t` for any other floating point value.

The value `-0.0` is distinguishable from ordinary zero in IEEE floating point, but Emacs Lisp `equal` and `=` consider them equal values.

You can use `logb` to extract the binary exponent of a floating point number (or estimate the logarithm of an integer):

**logb** *number* [Function]

This function returns the binary exponent of *number*. More precisely, the value is the logarithm of *number* base 2, rounded down to an integer.

```
(logb 10)
⇒ 3
(logb 10.0e20)
⇒ 69
```

### 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 31.2 [Predicates on Markers], page 573.

**floatp** *object* [Function]

This predicate tests whether its argument is a floating point number and returns `t` if so, `nil` otherwise.

`floatp` does not exist in Emacs versions 18 and earlier.

**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.

**wholenump** *object* [Function]

The `wholenump` predicate (whose name comes from the phrase “whole-number-p”) tests to see whether its argument is a nonnegative integer, and returns `t` if so, `nil` otherwise. 0 is considered non-negative.

`natnump` is an obsolete synonym for `wholenump`.

**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 `=`, not `eq`. There can be many distinct floating point number objects with the same numeric value. If you use `eq` to compare them, then you test whether two values are the same *object*. By contrast, `=` compares only the numeric values of the objects.

At present, each integer value has a unique Lisp object in Emacs Lisp. Therefore, `eq` is equivalent to `=` where integers are concerned. It is sometimes convenient to use `eq` for comparing an unknown value with an integer, 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 a good idea to use `=` if you can, even for comparing integers, just in case we change the representation of integers in a future Emacs version.

Sometimes it is useful to compare numbers with `equal`; it treats 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 2.7 [Equality Predicates], page 30.

There is another wrinkle: because floating point arithmetic is not exact, it is often a bad idea to check for equality of two 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 (and (= x 0) (= y 0))
      (< (/ (abs (- x y))
            (max (abs x) (abs y)))
         fuzz-factor)))
```

**Common Lisp note:** Comparing numbers in Common Lisp always requires `=` because Common Lisp implements multi-word integers, and two distinct integer objects can have the same numeric value. Emacs Lisp can have just one integer object for any given value because it has a limited range of integer values.

`= number-or-marker1 number-or-marker2` [Function]

This function tests whether its arguments are numerically equal, and returns `t` if so, `nil` otherwise.

`eq1 value1 value2` [Function]

This function acts like `eq` except when both arguments are numbers. It compares numbers by type and numeric value, so that `(eq1 1.0 1)` returns `nil`, but `(eq1 1.0 1.0)` and `(eq1 1 1)` both return `t`.

`/= 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-marker1 number-or-marker2` [Function]

This function tests whether its first argument is strictly less than its second argument. It returns `t` if so, `nil` otherwise.

**<=** *number-or-marker1* *number-or-marker2* [Function]

This function tests whether its first argument is less than or equal to its second argument. It returns **t** if so, **nil** otherwise.

**>** *number-or-marker1* *number-or-marker2* [Function]

This function tests whether its first argument is strictly greater than its second argument. It returns **t** if so, **nil** otherwise.

**>=** *number-or-marker1* *number-or-marker2* [Function]

This function tests whether its first argument is greater than or equal to its second 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. If any of the arguments is floating-point, the value is returned as floating point, even if it was given as an integer.

```
(max 20)
⇒ 20
(max 1 2.5)
⇒ 2.5
(max 1 3 2.5)
⇒ 3.0
```

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

This function returns the smallest of its arguments. If any of the arguments is floating-point, the value is returned as floating point, even if it was given as an integer.

```
(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 a floating point number, **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. An **arith-error** results if *divisor* is 0.

**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 may choose the integer closer to zero, or it may prefer an even integer, depending on your machine.

```
(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. Remainder and modulus functions supplement the division functions. The functions to add or subtract 1 are provided because they are traditional in Lisp and commonly used.

All of these functions except % return a floating point value if any argument is floating.

It is important to note that in Emacs Lisp, arithmetic functions do not check for overflow. Thus (1+ 268435455) may evaluate to -268435456, depending on your hardware.

### 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
```

**/ dividend divisor &rest divisors**

[Function]

This function divides *dividend* by *divisor* and returns the quotient. If there are additional arguments *divisors*, then it divides *dividend* by each divisor in turn. Each argument may be a number or a marker.

If all the arguments are integers, then the result is an integer too. This means the result has to be rounded. On most machines, the result is rounded towards zero after each division, but some machines may round differently with negative arguments. This is because the Lisp function / is implemented using the C division operator, which also permits machine-dependent rounding. As a practical matter, all known machines round in the standard fashion.

If you divide an integer by 0, an **arith-error** error is signaled. (See Section 10.5.3 [Errors], page 127.) Floating point division by zero returns either infinity or a NaN if your machine supports IEEE floating point; otherwise, it signals an **arith-error** error.

```
(/ 6 2) ⇒ 3
(/ 5 2) ⇒ 2
(/ 5.0 2) ⇒ 2.5
(/ 5 2.0) ⇒ 2.5
(/ 5.0 2.0) ⇒ 2.5
(/ 25 3 2) ⇒ 4
(/ -17 6) ⇒ -2    (could in theory be -3 on some machines)
```

**% dividend divisor**

[Function]

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

For negative arguments, the remainder is in principle machine-dependent since the quotient is; but in practice, all known machines behave alike.

An **arith-error** results if *divisor* is 0.

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

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

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

always equals *dividend*.

**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 returns a well-defined result for negative arguments. It also permits floating point arguments; it rounds the quotient downward (towards minus infinity) to an integer, and uses that quotient to compute the remainder.

An **arith-error** results if *divisor* is 0.

```
(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.

For floor, see Section 3.5 [Numeric Conversions], page 36.

## 3.7 Rounding Operations

The functions fffloor, fceiling, fround, and ftruncate take a floating point argument and return a floating point result whose value is a nearby integer. fffloor 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.

## 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). 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.

**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, **lsh** shifts zeros into the leftmost (most-significant) bit, producing a positive result even if *integer1* is negative. Contrast this with **ash**, below.

Here are two examples of **lsh**, shifting a pattern of bits one place to the left. We show only the low-order eight bits of the binary pattern; the rest are all zero.

```
(lsh 5 1)
      ⇒ 10
;; Decimal 5 becomes decimal 10.
00000101 ⇒ 00001010
```

```
(lsh 7 1)
      ⇒ 14
;; Decimal 7 becomes decimal 14.
00000111 ⇒ 00001110
```

As the examples illustrate, shifting the pattern of bits one place to the left produces a number that is twice the value of the previous number.

Shifting a pattern of bits two places to the left produces results like this (with 8-bit binary numbers):

```
(lsh 3 2)
      ⇒ 12
;; Decimal 3 becomes decimal 12.
00000011 ⇒ 00001100
```

On the other hand, shifting one place to the right looks like this:

```
(lsh 6 -1)
⇒ 3
;; Decimal 6 becomes decimal 3.
00000110 ⇒ 00000011
```

```
(lsh 5 -1)
⇒ 2
;; Decimal 5 becomes decimal 2.
00000101 ⇒ 00000010
```

As the example illustrates, shifting one place to the right divides the value of a positive integer by two, rounding downward.

The function `lsh`, like all Emacs Lisp arithmetic functions, does not check for overflow, so shifting left can discard significant bits and change the sign of the number. For example, left shifting 268,435,455 produces  $-2$  on a 29-bit machine:

```
(lsh 268435455 1) ; left shift
⇒ -2
```

In binary, in the 29-bit implementation, the argument looks like this:

```
;; Decimal 268,435,455
0 1111 1111 1111 1111 1111 1111 1111
```

which becomes the following when left shifted:

```
;; Decimal -2
1 1111 1111 1111 1111 1111 1111 1110
```

### `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.

`ash` gives the same results as `lsh` except when *integer1* and *count* are both negative. In that case, `ash` puts ones in the empty bit positions on the left, while `lsh` puts zeros in those bit positions.

Thus, with `ash`, shifting the pattern of bits one place to the right looks like this:

```
(ash -6 -1) ⇒ -3
;; Decimal -6 becomes decimal -3.
1 1111 1111 1111 1111 1111 1010
⇒
1 1111 1111 1111 1111 1111 1101
```

In contrast, shifting the pattern of bits one place to the right with `lsh` looks like this:

```
(lsh -6 -1) ⇒ 268435453
;; Decimal -6 becomes decimal 268,435,453.
1 1111 1111 1111 1111 1111 1010
⇒
0 1111 1111 1111 1111 1111 1101
```

Here are other examples:

```

;           29-bit binary values

(lsh 5 2)      ;   5  =  0 0000 0000 0000 0000 0000 0000 0101
                ;      =  0 0000 0000 0000 0000 0000 0001 0100
(ash 5 2)      ;   20
                ;      =  1 1111 1111 1111 1111 1111 1111 1011
                ;      =  1 1111 1111 1111 1111 1111 1110 1100
(ash -5 2)
                ;   -20
(lsh -5 2)     ;  -5  =  1 1111 1111 1111 1111 1111 1111 1011
                ;      =  1 1111 1111 1111 1111 1111 1110 1100
(ash -5 2)
                ;   -20
(lsh 5 -2)     ;   5  =  0 0000 0000 0000 0000 0000 0000 0101
                ;      =  0 0000 0000 0000 0000 0000 0000 0001
(ash 5 -2)
                ;   1
(lsh -5 -2)    ;  -5  =  1 1111 1111 1111 1111 1111 1111 1011
                ;      =  0 0111 1111 1111 1111 1111 1111 1110
(ash -5 -2)
                ;   -5  =  1 1111 1111 1111 1111 1111 1111 1011
                ;      =  1 1111 1111 1111 1111 1111 1111 1110
                ;      =  1 1111 1111 1111 1111 1111 1111 1110

```

**logand &rest ints-or-markers**

[Function]

This function returns the “logical and” of the arguments: the  $n$ th bit is set in the result if, and only if, the  $n$ th bit is set in all the arguments. (“Set” means that the value of the bit is 1 rather than 0.)

For example, using 4-bit binary numbers, the “logical and” of 13 and 12 is 12: 1101 combined with 1100 produces 1100. In both the binary numbers, the leftmost two bits are set (i.e., they are 1’s), so the leftmost two bits of the returned value are set. However, for the rightmost two bits, each is zero in at least one of the arguments, so the rightmost two bits of the returned value are 0’s.

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.

```

;           29-bit binary values

(logand 14 13)  ; 14  =  0 0000 0000 0000 0000 0000 0000 1110
                  ; 13  =  0 0000 0000 0000 0000 0000 0000 1101
                  ⇒ 12   ; 12  =  0 0000 0000 0000 0000 0000 0000 1100

(logand 14 13 4) ; 14  =  0 0000 0000 0000 0000 0000 0000 1110
                  ; 13  =  0 0000 0000 0000 0000 0000 0000 1101
                  ; 4   =  0 0000 0000 0000 0000 0000 0000 0100
                  ⇒ 4   ; 4   =  0 0000 0000 0000 0000 0000 0000 0100

(logand)
⇒ -1          ; -1  =  1 1111 1111 1111 1111 1111 1111 1111

```

**logior &rest ints-or-markers**

[Function]

This function returns the “inclusive or” of its arguments: the  $n$ th bit is set in the result if, and only if, the  $n$ th bit is set in at least one of the arguments. If there are no arguments, the result is zero, which is an identity element for this operation. If **logior** is passed just one argument, it returns that argument.

```
;           29-bit binary values

(logior 12 5) ; 12 = 0 0000 0000 0000 0000 0000 0000 1100
; 5 = 0 0000 0000 0000 0000 0000 0000 0101
⇒ 13 ; 13 = 0 0000 0000 0000 0000 0000 0000 1101

(logior 12 5 7) ; 12 = 0 0000 0000 0000 0000 0000 0000 1100
; 5 = 0 0000 0000 0000 0000 0000 0000 0101
; 7 = 0 0000 0000 0000 0000 0000 0000 0111
⇒ 15 ; 15 = 0 0000 0000 0000 0000 0000 0000 1111
```

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

This function returns the “exclusive or” of its arguments: the *n*th bit is set in the result if, and only if, the *n*th bit is set 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.

```
;           29-bit binary values

(logxor 12 5) ; 12 = 0 0000 0000 0000 0000 0000 0000 1100
; 5 = 0 0000 0000 0000 0000 0000 0000 0101
⇒ 9 ; 9 = 0 0000 0000 0000 0000 0000 0000 1001

(logxor 12 5 7) ; 12 = 0 0000 0000 0000 0000 0000 0000 1100
; 5 = 0 0000 0000 0000 0000 0000 0000 0101
; 7 = 0 0000 0000 0000 0000 0000 0000 0111
⇒ 14 ; 14 = 0 0000 0000 0000 0000 0000 0000 1110
```

**lognot integer** [Function]

This function returns the logical 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.

```
(lognot 5)
⇒ -6
;; 5 = 0 0000 0000 0000 0000 0000 0000 0101
;; becomes
;; -6 = 1 1111 1111 1111 1111 1111 1111 1010
```

## 3.9 Standard Mathematical Functions

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

<b>sin arg</b>	[Function]
<b>cos arg</b>	[Function]
<b>tan arg</b>	[Function]

These are the ordinary trigonometric functions, with argument measured in radians.

<b>asin arg</b>	[Function]
-----------------	------------

The value of (**asin arg**) is a number between  $-\pi/2$  and  $\pi/2$  (inclusive) whose sine is *arg*; if, however, *arg* is out of range (outside  $[-1, 1]$ ), it signals a **domain-error** error.

<b>acos arg</b>	[Function]
-----------------	------------

The value of (**acos arg**) is a number between 0 and  $\pi$  (inclusive) whose cosine is *arg*; if, however, *arg* is out of range (outside  $[-1, 1]$ ), it signals a **domain-error** error.

**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*. *e* is a fundamental mathematical constant also called the base of natural logarithms.

**log** *arg* &**optional** *base* [Function]

This function returns the logarithm of *arg*, with base *base*. If you don't specify *base*, the base *e* is used. If *arg* is negative, it signals a **domain-error** error.

**log10** *arg* [Function]

This function returns the logarithm of *arg*, with base 10. If *arg* is negative, it signals a **domain-error** error. (**log10** *x*)  $\equiv$  (**log** *x* 10), at least approximately.

**expt** *x* *y* [Function]

This function returns *x* raised to power *y*. If both arguments are integers and *y* is positive, the result is an integer; in this case, overflow causes truncation, so watch out.

**sqrt** *arg* [Function]

This returns the square root of *arg*. If *arg* is negative, it signals a **domain-error** error.

## 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.

In Emacs, pseudo-random numbers are generated from a “seed” number. Starting from any given seed, the **random** function always generates the same sequence of numbers. Emacs always starts with the same seed value, so the sequence of values of **random** is actually the same in each Emacs run! For example, in one operating system, the first call to (**random**) after you start Emacs always returns  $-1457731$ , and the second one always returns  $-7692030$ . This repeatability is helpful for debugging.

If you want random numbers that don't always come out the same, execute (**random** *t*). This chooses a new seed based on the current time of day and on Emacs's process ID number.

**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*.

If *limit* is *t*, it means to choose a new seed based on the current time of day and on Emacs's process ID number.

On some machines, any integer representable in Lisp may be the result of `random`.  
On other machines, the result can never be larger than a certain maximum or less  
than a certain (negative) minimum.

## 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 21.6.14 [Strings of Events], page 328, for special considerations for strings of keyboard character events.

### 4.1 String and Character Basics

Characters are represented in Emacs Lisp as integers; whether an integer is a character or not is determined only by how it is used. Thus, strings really contain integers.

The length of a string (like any array) is fixed, and cannot be altered once the string exists. Strings in Lisp are *not* terminated by a distinguished character code. (By contrast, strings in C are terminated by a character with ASCII code 0.)

Since strings are arrays, and therefore sequences as well, you can operate on them with the general array and sequence functions. (See Chapter 6 [Sequences Arrays Vectors], page 87.) For example, you can access or change individual characters in a string using the functions `aref` and `aset` (see Section 6.3 [Array Functions], page 90).

There are two text representations for non-ASCII characters in Emacs strings (and in buffers): unibyte and multibyte (see Section 33.1 [Text Representations], page 640). An ASCII character always occupies one byte in a string; in fact, when a string is all ASCII, there is no real difference between the unibyte and multibyte representations. For most Lisp programming, you don't need to be concerned with these two representations.

Sometimes key sequences are represented as strings. When a 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 2.3.3 [Character Type], page 10, for more information about the representation of meta and other modifiers for keyboard input characters.

Strings are useful for holding regular expressions. You can also match regular expressions against strings with `string-match` (see Section 34.4 [Regexp Search], page 673). The functions `match-string` (see Section 34.6.2 [Simple Match Data], page 677) and `replace-match` (see Section 34.6.1 [Replacing Match], page 676) 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 32.19 [Text Properties], page 615. 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 32 [Text], page 581, for information about functions that display strings or copy them into buffers. See Section 2.3.3 [Character Type], page 10, and Section 2.3.8

[String Type], page 18, for information about the syntax of characters and strings. See Chapter 33 [Non-ASCII Characters], page 640, for functions to convert between text representations and to encode and decode character codes.

## 4.2 The Predicates for Strings

For more information about general sequence and array predicates, see Chapter 6 [Sequences Arrays Vectors], page 87, and Section 6.2 [Arrays], page 89.

**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`, `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.

**make-string** *count character* [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)
⇒ "
```

Other functions to compare with this one include **char-to-string** (see Section 4.6 [String Conversion], page 54), **make-vector** (see Section 6.4 [Vectors], page 91), and **make-list** (see Section 5.4 [Building Lists], page 67).

**string &rest characters** [Function]

This returns a string containing the characters *characters*.

```
(string ?a ?b ?c)
⇒ "abc"
```

**substring** *string start &optional 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.

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

Here the index for ‘a’ is 0, the index for ‘b’ is 1, and the index for ‘c’ is 2. Thus, three letters, ‘abc’, are copied from the string “abcdefg”. The index 3 marks the

character position up to which the substring is copied. The character whose index is 3 is actually the fourth character in the string.

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 87).

If the characters copied from `string` have text properties, the properties are copied into the new string also. See Section 32.19 [Text Properties], page 615.

`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 32.2 [Buffer Contents], page 582), 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 new 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)
⇒ ""
```

The `concat` function always constructs a new string that is not `eq` to any existing string.

In Emacs versions before 21, when an argument was an integer (not a sequence of integers), it was converted to a string of digits making up the decimal printed representation of the integer. This obsolete usage no longer works. The proper way to convert an integer to its decimal printed form is with `format` (see Section 4.7 [Formatting Strings], page 56) or `number-to-string` (see Section 4.6 [String Conversion], page 54).

For information about other concatenation functions, see the description of `mapconcat` in Section 12.6 [Mapping Functions], page 168, `vconcat` in Section 6.5 [Vector Functions], page 92, and `append` in Section 5.4 [Building Lists], page 67.

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

This function splits *string* into substrings at matches for the regular expression *separators*. Each match for *separators* defines a splitting point; the substrings between the splitting points are made into a list, which is the value returned by `split-string`.

If *omit-nulls* is `nil`, 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 *separators* is `nil` (or omitted), the default is the value of `split-string-default-separators`.

As a special case, when *separators* is `nil` (or omitted), null strings are always omitted from the result. Thus:

```
(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" "")
```

More examples:

```
(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 "aoob" "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`".

## 4.4 Modifying Strings

The most basic way to alter the contents of an existing string is with `aset` (see Section 6.3 [Array Functions], page 90). (`aset string idx char`) stores `char` into `string` at index `idx`. Each character occupies one or more bytes, and if `char` needs a different number of bytes from the character already present at that index, `aset` signals an error.

A more powerful function is `store-substring`:

**store-substring** `string idx obj` [Function]  
 This function alters part of the contents of the string `string`, by storing `obj` starting at index `idx`. The argument `obj` may be either a character or a (smaller) string.

Since it is impossible to change the length of an existing string, it is an error if `obj` doesn’t fit within `string`’s actual length, or if any new character requires a different number of bytes from the character currently present at that point in `string`.

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 their print names are used. Case is always significant, regardless of **case-fold-search**.

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

The function **string=** ignores the text properties of the two strings. When **equal** (see Section 2.7 [Equality Predicates], page 30) compares two strings, it uses **string=**.

For technical reasons, a unibyte and a multibyte string are **equal** if and only if they contain the same sequence of character codes and all these codes are either in the range 0 through 127 (ASCII) or 160 through 255 (**eight-bit-graphic**). However, when a unibyte string gets converted to a multibyte string, all characters with codes in the range 160 through 255 get converted to characters with higher codes, whereas ASCII characters remain unchanged. Thus, a unibyte string and its conversion to multibyte are only **equal** if the string is all ASCII. Character codes 160 through 255 are not entirely proper in multibyte text, even though they can occur. As a consequence, the situation where a unibyte and a multibyte string are **equal** without both being all ASCII is a technical oddity that very few Emacs Lisp programmers ever get confronted with. See Section 33.1 [Text Representations], page 640.

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

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

**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 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 33.1 [Text Representations], page 640).

```
(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 used.

**string-lessp** `string1` `string2` [Function]  
`string-lessp` is another name for `string<`.

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

This function compares the specified part of `string1` with the specified part of `string2`. The specified part of `string1` runs from index `start1` up to index `end1` (`nil` means the end of the string). The specified part of `string2` runs from index `start2` up to index `end2` (`nil` means the end of the string).

The strings are both converted to multibyte for the comparison (see Section 33.1 [Text Representations], page 640) so that a unibyte string and its conversion to multibyte are always regarded as equal. If `ignore-case` is non-`nil`, then case is ignored, so that upper case letters can be equal to lower case letters.

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.

**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*, it ignores case differences. 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 81.

See also the **compare-buffer-substrings** function in Section 32.3 [Comparing Text], page 584, 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 34.4 [Regexp Search], page 673.

## 4.6 Conversion of Characters and Strings

This section describes functions for conversions between characters, strings and integers. **format** (see Section 4.7 [Formatting Strings], page 56) and **prin1-to-string** (see Section 19.5 [Output Functions], page 273) can also convert Lisp objects into strings. **read-from-string** (see Section 19.3 [Input Functions], page 271) can “convert” a string representation of a Lisp object into an object. The functions **string-make-multibyte** and **string-make-unibyte** convert the text representation of a string (see Section 33.2 [Converting Representations], page 641).

See Chapter 24 [Documentation], page 425, 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.

**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 48.

**string-to-char** *string* [Function]

This function returns the first character in *string*. If the string is empty, the function returns 0. The value is also 0 when the first character of *string* is the null character, ASCII code 0.

```
(string-to-char "ABC")
⇒ 65

(string-to-char "xyz")
⇒ 120
(string-to-char "")
⇒ 0
(string-to-char "\000")
⇒ 0
```

This function may be eliminated in the future if it does not seem useful enough to retain.

**number-to-string** *number* [Function]

This function returns a string consisting of the printed base-ten representation of *number*, which may be an integer or a floating point 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 56.

**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. If *string* looks like an integer but its value is too large to fit into a Lisp integer, **string-to-number** returns a floating point result.

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 the first character after the ignored whitespace is neither a digit in the given base, nor a plus or minus sign, nor the leading dot of a floating point 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.

Here are some other functions that can convert to or from a string:

**concat** **concat** can convert a vector or a list into a string. See Section 4.3 [Creating Strings], page 48.

**vconcat** **vconcat** can convert a string into a vector. See Section 6.5 [Vector Functions], page 92.

**append** **append** can convert a string into a list. See Section 5.4 [Building Lists], page 67.

## 4.7 Formatting Strings

*Formatting* means constructing a string by substitution of 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` only in how they use the result of formatting.

### `format string &rest objects`

[Function]

This function returns a new string that is made by copying *string* and then replacing any format specification in the copy 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.

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)
```

If *string* contains more than one format specification, the 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.

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 <code>princ</code> , not <code>prin1</code> —see Section 19.5 [Output Functions], page 273). 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 <code>prin1</code> —see Section 19.5 [Output Functions], page 273). 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.

'%d'	Replace the specification with the base-ten representation of an integer.
'%x'	
'%X'	Replace the specification with the base-sixteen representation of an integer. '%x' uses lower case and '%X' uses upper case.
'%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, whichever is shorter.
'%%'	Replace the specification with a single '%'. This format specification is unusual in 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:

```
(format "The name of this buffer is %s." (buffer-name))
⇒ "The name of this buffer is strings.texi."

(format "The buffer object prints as %s." (current-buffer))
⇒ "The buffer object prints as strings.texi."

(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."
```

A specification can have a *width*, which is a signed decimal number between the '%' and the specification character. If the printed representation of the object contains fewer characters than this width, `format` extends it with padding. The padding goes on the left if the width is positive (or starts with zero) and on the right if the width is negative. The padding character is normally a space, but it’s ‘0’ if the width starts with a zero.

Some of these conventions are ignored for specification characters for which they do not make sense. That is, ‘%s’, ‘%S’ and ‘%c’ accept a width starting with 0, but still pad with *spaces* on the left. Also, ‘%%’ accepts a width, but ignores it. Here are some examples of padding:

```
(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"
```

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 three examples, ‘%7s’ specifies a minimum width of 7. In the first case, the string inserted in place of ‘%7s’ has only 3 letters, it needs 4 blank spaces as padding. In the second case, the string “specification” is 13 letters wide but is not truncated. In the third case, the padding is 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."

(format "The word '%7s' actually has %d letters in it."
       "specification" (length "specification"))
⇒ "The word 'specification' actually has 13 letters in it.

(format "The word '%-7s' actually has %d letters in it."
       "foo" (length "foo"))
⇒ "The word 'foo      ' actually has 3 letters in it."
```

All the specification characters allow an optional *precision* before the character (after the width, if present). The precision is a decimal-point ‘.’ followed by a digit-string. For the floating-point specifications (‘%e’, ‘%f’, ‘%g’), the precision specifies how many decimal places to show; if zero, the decimal-point itself is also omitted. 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*. Precision has no effect for other specification characters.

Immediately after the ‘%’ and before the optional width and precision, you can put certain “flag” characters.

‘+’ as a flag inserts a plus sign before a positive number, so that it always has a sign. A space character as flag inserts a space before a positive number. (Otherwise, positive numbers start with the first digit.) Either of these two flags ensures that positive numbers and negative numbers use the same number of columns. These flags are ignored except for ‘%d’, ‘%e’, ‘%f’, ‘%g’, and if both flags are used, the ‘+’ 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 the result with ‘0x’ or ‘0X’. For ‘%e’, ‘%f’, and ‘%g’, the ‘#’ flag means include a decimal point even if the precision is zero.

## 4.8 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 4.9 [Case Tables], page 60).

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 a character or a string to lower case.

When the argument to **downcase** is a string, the function creates and returns a new string in which each letter in the argument that is upper case is converted to lower

case. When the argument to `downcase` is a character, `downcase` returns the corresponding lower case character. This value is an integer. If the original character is lower case, or is not a letter, then the value equals 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 a character or a string to upper case.

When the argument to `upcase` is a string, the function creates and returns a new string in which each letter in the argument that is lower case is converted to upper case.

When the argument to `upcase` is a character, `upcase` returns the corresponding upper case character. This value is an integer. If the original character is upper case, or is not a letter, then the value returned equals 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 creates and 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 35.2.1 [Syntax Class Table], page 685).

When the argument to `capitalize` is a character, `capitalize` has the same result 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 35.2.1 [Syntax Class Table], page 685).

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"
```

See Section 4.5 [Text Comparison], page 52, for functions that compare strings; some of them ignore case differences, or can optionally ignore case differences.

## 4.9 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 32.18 [Case Changes], page 613). 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 93) 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 you construct 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 10.5 [Nonlocal Exits], page 125).

Some language environments may modify the case conversions of ASCII characters; for example, in the Turkish language environment, the ASCII character ‘I’ is downcased into a Turkish “dotless i”. 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 35 [Syntax Tables], page 684. 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*. 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 2.3.6 [Cons Cell Type], page 14.) 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: the CDR slot of each cons cell refers to the following cons cell. The CDR of the last cons cell is `nil`. This asymmetry between the CAR and the CDR is entirely a matter of convention; at the level of cons cells, the CAR and CDR slots have the same characteristics.

Since `nil` is the conventional value to put in the CDR of the last cons cell in the list, we call that case a *true list*.

In Lisp, we consider the symbol `nil` a list as well as a symbol; it is the list with no elements. For convenience, the symbol `nil` is considered to have `nil` as its CDR (and also as its CAR). Therefore, the CDR of a true list is always a true list.

If the CDR of a list’s last cons cell is some other value, neither `nil` nor another cons cell, we call the structure a *dotted list*, since its printed representation would use ‘.’. 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 true, circular or dotted. If the 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 true 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.

Because most cons cells are used as part of lists, the phrase *list structure* has come to mean any structure made out of cons cells.

The CDR of any nonempty true list *l* is a list containing all the elements of *l* except the first.

See Section 2.3.6 [Cons Cell Type], page 14, for the read and print syntax of cons cells and lists, and for “box and arrow” illustrations of lists.

## 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 all of 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 `null` when *object* is considered a list and `not` when it is considered a truth value (see `not` in Section 10.3 [Combining Conditions], page 122).

```
(null '(1))
⇒ nil
(null '())
⇒ t
```

## 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*. Expressed another way, this function returns the CAR of *cons-cell*.

As a special case, if *cons-cell* is `nil`, then `car` is defined to return `nil`; therefore, any list is a valid argument for `car`. 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*. Expressed another way, this function returns the CDR of *cons-cell*.

As a special case, if *cons-cell* is **nil**, then **cdr** is defined to return **nil**; therefore, any list is a valid argument for **cdr**. 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 is a way of examining the CAR of a list, and taking it off the list, all at once.

It operates on the list which is stored in the symbol *listname*. It removes this element from the list by setting *listname* to the CDR of its old value—but it also returns the CAR of that list, which is the element being removed.

```
x
  ⇒ (a b c)
(pop x)
```

```

⇒ a
x
⇒ (b c)

```

**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**.

If *n* is negative, **nth** returns the first element of *list*.

```

(nth 2 '(1 2 3 4))
⇒ 3
(nth 10 '(1 2 3 4))
⇒ nil
(nth -3 '(1 2 3 4))
⇒ 1

```

```
(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 87.

**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 or negative, **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 -3 '(1 2 3 4))
⇒ (1 2 3 4)

```

**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 87.

<b>caar</b> cons-cell	[Function]
This is the same as (car (car cons-cell)).	
<b>cadr</b> cons-cell	[Function]
This is the same as (car (cdr cons-cell)) or (nth 1 cons-cell).	
<b>cdar</b> cons-cell	[Function]
This is the same as (cdr (car cons-cell)).	
<b>cddr</b> cons-cell	[Function]
This is the same as (cdr (cdr cons-cell)) or (nthcdr 2 cons-cell).	
<b>butlast</b> 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.	
<b>nbutlast</b> 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**.

<b>cons</b> object1 object2	[Function]
This function is the most basic function for building new list structure. It creates a new cons cell, making <i>object1</i> the CAR, and <i>object2</i> the CDR. It then returns the new cons cell. The arguments <i>object1</i> and <i>object2</i> may be any Lisp objects, but most often <i>object2</i> 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*.<sup>1</sup> 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.

---

<sup>1</sup> 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.

**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 48).

```
(make-list 3 'pigs)
      ⇒ (pigs pigs pigs)
(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 75, 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 true list.

In Emacs 20 and before, the **append** function also allowed integers as (non last) arguments. It converted them to strings of digits, making up the decimal print representation of the integer, and then used the strings instead of the original integers. This obsolete usage no longer works. The proper way to convert an integer to a decimal number in this way is with **format** (see Section 4.7 [Formatting Strings], page 56) or **number-to-string** (see Section 4.6 [String Conversion], page 54).

Here is an example of using **append**:

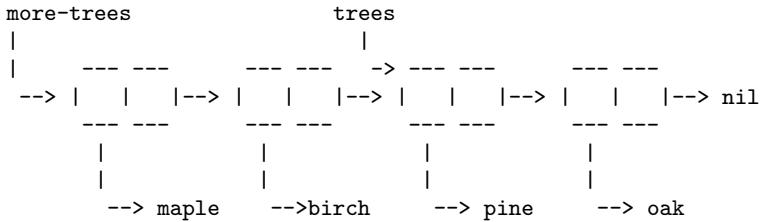
```
(setq trees '(pine oak))
      ⇒ (pine oak)
(setq more-trees (append '(maple birch) trees))
      ⇒ (maple birch pine oak)
```

```

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 87.

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 12.5 [Calling Functions], page 167), 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.

**reverse** *list* [Function]

This function creates a new list whose elements are the elements of *list*, but in reverse order. The original argument *list* is *not* altered.

```
(setq x '(1 2 3 4))
  ⇒ (1 2 3 4)
(reverse x)
  ⇒ (4 3 2 1)
x
  ⇒ (1 2 3 4)
```

**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).

**number-sequence** *from* &optional *to* *separation* [Function]

This 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 can be integers or floating point numbers. However, 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 newelt listname** [Macro]

This macro provides an alternative way to write `(setq listname (cons newelt listname))`.

```
(setq l '(a b))
  ⇒ (a b)
(push 'c l)
  ⇒ (c a b)
l
  ⇒ (c a b)
```

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.

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.
```

$\Rightarrow (c \ a \ b)$

An equivalent expression for (add-to-list 'var value) is this:

```
(or (member value 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 that is what you want.

The ordering information is stored in a hash table on `symbol`'s `list-order` property.

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`. We call these “destructive” operations because they change existing list structure.

**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 '(1 2))
      ⇒ (1 2)
(setcar x 4)
      ⇒ 4
x
      ⇒ (4 2)
```

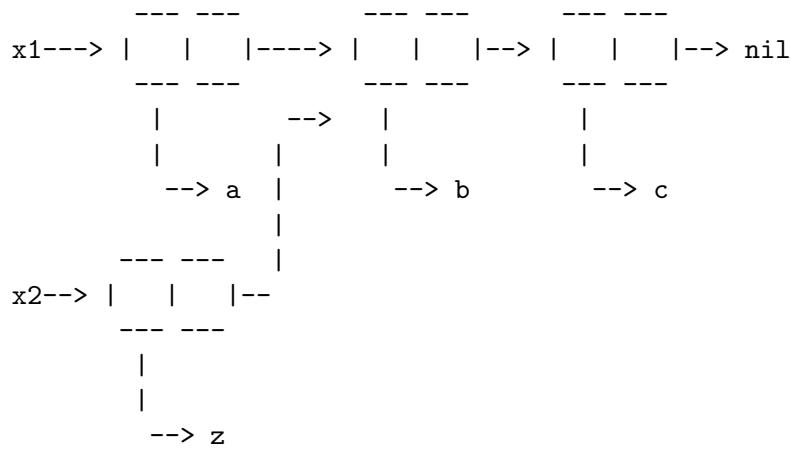
When a cons cell is part of the shared structure of several lists, storing a new CAR into the cons changes one element of each of these lists. Here is an example:

```
; ; Create two lists that are partly shared.
(setq x1 '(a b c))
      ⇒ (a b c)
(setq x2 (cons 'z (cdr x1)))
      ⇒ (z b c)

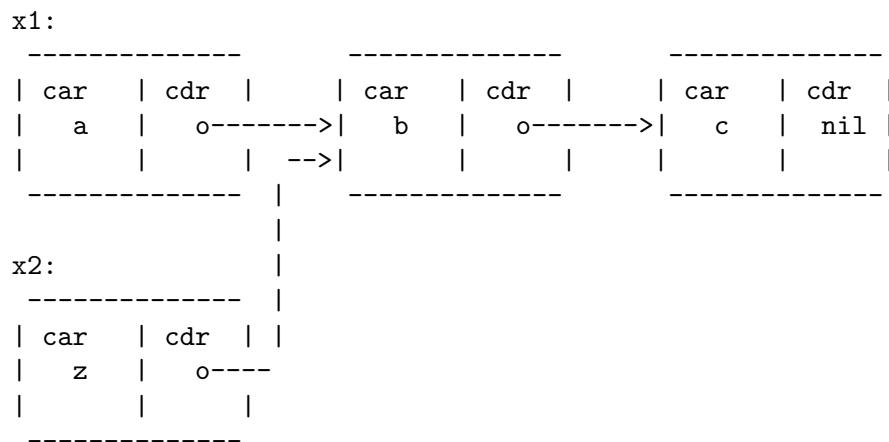
; ; Replace the CAR of a shared link.
(setcar (cdr x1) 'foo)
      ⇒ foo
x1
      ⇒ (a foo c)
x2
      ⇒ (z foo c)

; ; Replace the CAR of a link that is not shared.
(setcar x1 'baz)
      ⇒ baz
x1
      ⇒ (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*.

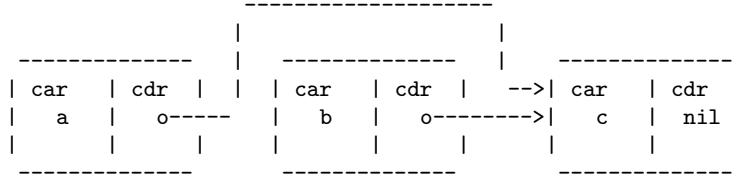
Here is an example of replacing the CDR of a list with a different list. All but the first element of the list are removed in favor of a different sequence of elements. The first element is unchanged, because it resides in the CAR of the list, and is not reached via the CDR.

```
(setq x '(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 '(a b c))
  ⇒ (a b c)
(setcdr x1 (cdr (cdr x1)))
  ⇒ (c)
x1
  ⇒ (a c)
```

Here is the result in box notation:

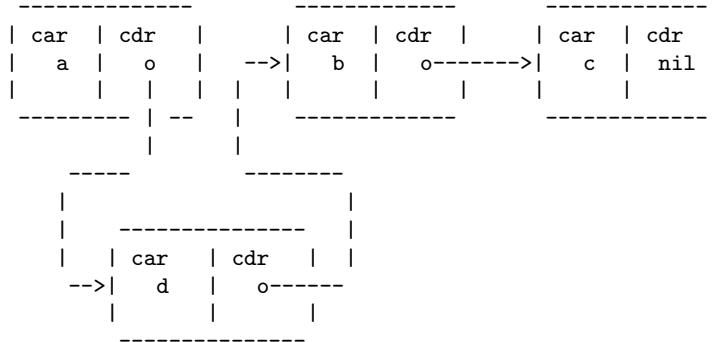


The second cons cell, which previously held the element b, still exists and its CAR is still b, but it no longer forms part of this list.

It is equally easy to insert a new element by changing CDRs:

```
(setq x1 '(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. We call these functions “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 67), 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 '(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 '(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) must be lists.

A common pitfall is to use a quoted constant list as a non-last argument to **nconc**. If you do this, your program will change each time you run it! Here is what happens:

```
(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 (quote (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 (quote (foo 1 2 3 4)) x)))
```

**nreverse** *list*

[Function]

This function reverses the order of the elements of *list*. Unlike **reverse**, **nreverse** alters its argument by reversing the CDRs in the cons cells forming the list. The cons cell that used to be the last one in *list* becomes the first cons cell of the value.

For example:

```
(setq x '(a b c))
  ⇒ (a b c)
```

```

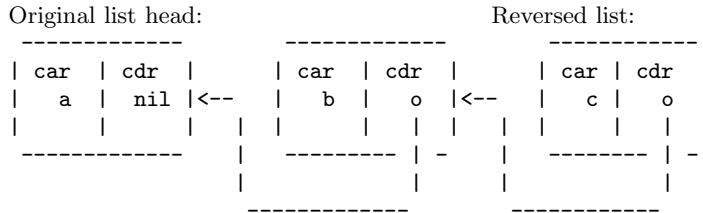
x
  ⇒ (a b c)
(nreverse x)
  ⇒ (c b a)
;; The cons cell that was first is now last.
x
  ⇒ (a)

```

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:



**sort** *list predicate* [Function]

This function sorts *list* stably, though destructively, and returns the sorted list. It 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 *list*. 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` is that it rearranges the cons cells forming *list* by changing CDRS. A nondestructive sort function would create new cons cells to store the elements in their sorted order. If you wish to make a sorted copy without destroying the original, copy it first with `copy-sequence` and then `sort`.

Sorting does not change the CARS of the cons cells in *list*; the cons cell that originally contained the element *a* in *list* still has *a* in its CAR after sorting, but it now appears in a different position in the list due to the change of CDRS. For example:

```

(setq nums '(1 3 2 6 5 4 0))
  ⇒ (1 3 2 6 5 4 0)
(sort nums '<)
  ⇒ (0 1 2 3 4 5 6)
nums
  ⇒ (1 2 3 4 5 6)

```

**Warning:** Note that the list in `nums` no longer contains 0; this is the same cons cell that it was before, but it is no longer the first one in the list. Don't assume a variable that formerly held the argument now holds the entire sorted list! Instead, save the result of `sort` and use that. Most often we store the result back into the variable that held the original list:

```
(setq nums (sort nums '<))
```

See Section 32.15 [Sorting], page 605, for more functions that perform sorting. See documentation in Section 24.2 [Accessing Documentation], page 426, for a useful example of `sort`.

## 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`. 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, but GNU Emacs Lisp does not have them. You can write them in Lisp if you wish.

`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))) ; (2) and (2) are not eq.
      ⇒ nil
```

`delq object list`

[Function]

This function destructively removes all elements `eq` to `object` from `list`. The letter 'q' in `delq` says that it uses `eq` to compare `object` against the elements of the list, like `memq` and `remq`.

When `delq` deletes elements from the front of the list, it does so simply by advancing down the list and returning a sublist that starts after those elements:

```
(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 74).

```
(setq sample-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 (4) that `delq` attempts to match and the (4) in the `sample-list` are not `eq`:

```
(delq '(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 '(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)) ; 1.2 and 1.2 are not eq.
  ⇒ nil
```

The following three functions are like `memq`, `delq` and `remq`, but use `equal` rather than `eq` to compare elements. See Section 2.7 [Equality Predicates], page 30.

### `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))) ; (2) and (2) are not eq.
⇒ nil
;; Two strings with the same contents are equal.
(member "foo" ("foo" "bar"))
⇒ ("foo" "bar")
```

### `delete object sequence`

[Function]

If `sequence` is a list, this function destructively removes all elements `equal` to `object` from `sequence`. For lists, `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.

If `sequence` is a vector or string, `delete` returns a copy of `sequence` with all elements `equal` to `object` removed.

For example:

```
(setq l '((2) (1) (2)))
(delete '(2) l)
⇒ ((1))
l
⇒ ((2) (1))
;; If you want to change l reliably,
;; write (setq l (delete elt l)).
(setq l '((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 also the function **add-to-list**, in Section 5.5 [List Variables], page 71, 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*.<sup>2</sup>

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.

---

<sup>2</sup> 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.

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 8.4 [Property Lists], page 107, for a comparison of property lists and association lists.

**assoc** *key alist*

[Function]

This function returns the first association for *key* in *alist*, comparing *key* against the *alist* elements using **equal** (see Section 2.7 [Equality Predicates], page 30). It returns **nil** if no association in *alist* has a CAR **equal** to *key*. For example:

```
(setq trees '((pine . cones) (oak . acorns) (maple . seeds)))
      ⇒ ((pine . cones) (oak . acorns) (maple . seeds))
(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 52.

**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** instead of **equal**. **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 2.7 [Equality Predicates], page 30.

```
(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)
⇒ nil
(assoc "simple leaves" leaves)
⇒ ("simple leaves" . oak)

```

**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 34.4 [Regexp Search], page 673). 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 '((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))
```

### `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.

## 5.9 Managing a Fixed-Size Ring of Objects

This section describes functions for operating on rings. A *ring* is a fixed-size data structure that supports insertion, deletion, rotation, and modulo-indexed reference and traversal.

### `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.

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")
```

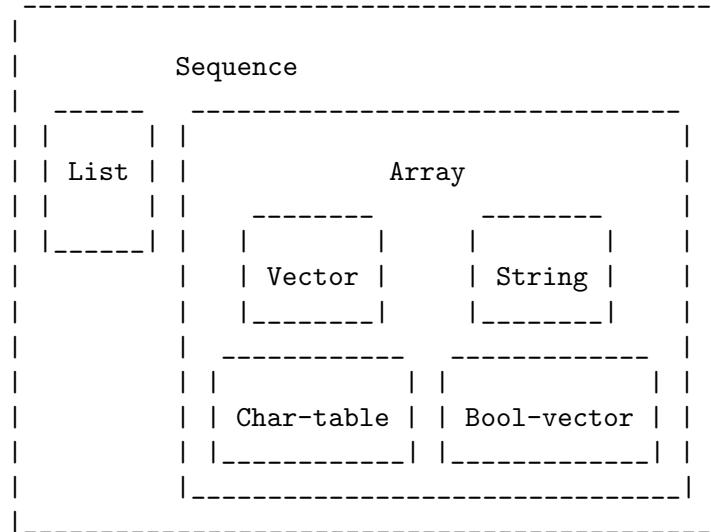
## 6 Sequences, Arrays, and Vectors

Recall that 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 single primitive object that has a slot for each of its elements. All the elements are accessible in constant time, but the length of an existing array cannot be changed. Strings, vectors, char-tables and bool-vectors are the four types of arrays.

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:



The elements of vectors and lists may be any Lisp objects. The elements of strings are all characters.

### 6.1 Sequences

In Emacs Lisp, a sequence is either a list or an array. The common property of all sequences is that they are ordered collections of elements. This section describes functions that accept any kind of sequence.

**sequencep object** [Function]

Returns `t` if *object* is a list, vector, string, bool-vector, or char-table, `nil` otherwise.

**length sequence** [Function]

This function returns the number of elements in *sequence*. If *sequence* is a dotted list, a **wrong-type-argument** error is signaled. Circular lists may cause an infinite loop. For a char-table, the value returned is always one more than the maximum Emacs character code.

See [Definition of `safe-length`], page 66, 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 33.1 [Text Representations], page 640.

**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 66. 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 90) and `nth` (see [Definition of `nth`], page 66).

**copy-sequence sequence** [Function]

Returns a copy of *sequence*. The copy is the same type of object as the original sequence, and it has the same elements in the same order.

Storing a new element into the copy does not affect the original *sequence*, and vice versa. However, the elements of the new sequence are not copies; they are identical (`eq`) to the elements of the original. Therefore, changes made within these elements, as found via the copied sequence, are also visible in the original sequence.

If the sequence 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 32.19 [Text Properties], page 615.

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 67, `concat` in Section 4.3 [Creating Strings], page 48, and `vconcat` in Section 6.5 [Vector Functions], page 92, for other ways to copy sequences.

```

(setq bar '(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)]

```

## 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, an element of a list requires access time that is proportional to the position of the element in the list.

Emacs defines four types of array, all one-dimensional: *strings*, *vectors*, *bool-vectors* and *char-tables*. A vector is a general array; its elements can be any Lisp objects. A string is a specialized array; its elements must be characters. Each type of array has its own read syntax. See Section 2.3.8 [String Type], page 18, and Section 2.3.9 [Vector Type], page 20.

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—in other words, 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 90).

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 32.19 [Text Properties], page 615.
- 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 47.

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 21.7.1 [Key Sequence Input], page 330.

## 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** *array index*

[Function]

This function returns the *index*th element of *array*. 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 87.

### **aset** *array index object*

[Function]

This function sets the *index*th element of *array* to be *object*. It returns *object*.

```
(setq w [foo bar baz])
⇒ [foo bar baz]
(aset w 0 'fu)
⇒ fu
w
⇒ [fu bar baz]
```

```
(setq x "asdfasfd")
  ⇒ "asdfasfd"
(aset x 3 ?Z)
  ⇒ 90
x
  ⇒ "asdZasfd"
```

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 [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 "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 87.

## 6.4 Vectors

Arrays in Lisp, like arrays in most languages, are blocks of memory whose elements can be accessed in constant time. A vector is a general-purpose array of specified length; its elements can be any Lisp objects. (By contrast, a string can hold only characters as elements.) Vectors in Emacs are used for obarrays (vectors of symbols), and as part of keymaps (vectors of commands). They are also used internally as part of the representation of a byte-compiled function; if you print such a function, you will see a vector in it.

In Emacs Lisp, the indices of the elements of a vector start from zero and count up from there.

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 9.1.1 [Self-Evaluating Forms], page 111.

Here are examples illustrating these principles:

```
(setq avector [1 two '(three) "four" [five]])
  ⇒ [1 two (quote (three)) "four" [five]]
(eval avector)
  ⇒ [1 two (quote (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 the *sequences*. The arguments *sequences* may be true lists, vectors, strings or bool-vectors. If no *sequences* are given, an empty vector is returned.

The value is a newly constructed 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 16.7 [Byte-Code Objects], page 220.

In Emacs versions before 21, the `vconcat` function allowed integers as arguments, converting them to strings of digits, but that feature has been eliminated. The proper way to convert an integer to a decimal number in this way is with `format` (see Section 4.7 [Formatting Strings], page 56) or `number-to-string` (see Section 4.6 [String Conversion], page 54).

For other concatenation functions, see `mapconcat` in Section 12.6 [Mapping Functions], page 168, `concat` in Section 4.3 [Creating Strings], page 48, and `append` in Section 5.4 [Building Lists], page 67.

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 (quote (three)) "four" [five]]
(append avector nil)
  ⇒ (1 two (quote (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. Char-tables are constants when evaluated.

Each char-table has a *subtype* which is a symbol. The subtype has two purposes: to distinguish char-tables meant for different uses, and to control the number of extra slots. For example, display tables are char-tables with `display-table` as the subtype, and syntax tables are char-tables with `syntax-table` as the subtype. A valid subtype must have a `char-table-extra-slots` property which is an integer between 0 and 10. This integer specifies the number of *extra slots* in the char-table.

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*. 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.

**char-table-p** *object* [Function]

This function returns `t` if *object* is a char-table, otherwise `nil`.

**char-table-subtype** *char-table* [Function]

This function returns the subtype symbol of *char-table*.

**set-char-table-default** *char-table char new-default* [Function]

This function sets the default value of generic character *char* in *char-table* to *new-default*.

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* 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* 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).

**charset** Refers to the value specified for the whole character set *charset* (see Section 33.5 [Character Sets], page 644).

**generic-char**

A generic character stands for a character set, or a row of a character set; specifying the generic character as argument is equivalent to specifying the character set name. See Section 33.7 [Splitting Characters], page 645, for a description of generic characters.

**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).

**charset** Refers to the value specified for the whole character set *charset* (see Section 33.5 [Character Sets], page 644).

**generic-char**

A generic character stands for a character set; specifying the generic character as argument is equivalent to specifying the character set name. See Section 33.7 [Splitting Characters], page 645, for a description of generic characters.

**map-char-table** *function char-table*

[Function]

This function calls *function* for each element of *char-table*. *function* is called 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 generic character—and the value is (**char-table-range** *char-table* *key*).

Overall, the key-value pairs passed to *function* describe all the values stored in *char-table*.

The return value is always **nil**; to make this function useful, *function* should have side effects. For example, here is how to examine each element of the syntax table:

```
(let (accumulator)
  (map-char-table
    #'(lambda (key value)
        (setq accumulator
              (cons (list key value) accumulator)))
    (syntax-table))
  accumulator)
⇒
((475008 nil) (474880 nil) (474752 nil) (474624 nil)
 ... (5 (3)) (4 (3)) (3 (3)) (2 (3)) (1 (3)) (0 (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.

There are two special functions for working 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-p** *object*

[Function]

This returns **t** if *object* is a bool-vector, and **nil** otherwise.

Here is an example of creating, examining, and updating a bool-vector. Note that the printed form represents up to 8 boolean values as a single character.

```
(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.

## 7 Hash Tables

A hash table is a very fast kind of lookup table, somewhat like an alist (see Section 5.8 [Association Lists], page 81) 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 no read syntax, and print in hash notation, like this:

```
(make-hash-table)
⇒ #<hash-table 'eql nil 0/65 0x83af980>
```

(The term “hash notation” refers to the initial ‘#’ character—see Section 2.1 [Printed Representation], page 8—and has nothing to do with the term “hash table.”)

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 8.3 [Creating Symbols], page 104).

### 7.1 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 numbers; 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 7.3 [Defining Hash], page 99) 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 that much to the nominal size. If `rehash-size` is a floating point number, it had better be greater than 1, and the hash table grows by multiplying the old size by 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 this fraction of the nominal size. The default for `threshold` is 0.8.

**makehash &optional test** [Function]

This is equivalent to `make-hash-table`, but with a different style argument list. The argument `test` specifies the method of key lookup.

This function is obsolete. Use `make-hash-table` instead.

## 7.2 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.

**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.

**Common Lisp note:** In Common Lisp, `clrhash` returns the empty `table`. In Emacs Lisp, it returns `nil`.

**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`.

## 7.3 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 reduces 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.

**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 integer values for hash codes, including negative integers.

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** *obj* [Function]

This function returns a hash code for Lisp object *obj*. This is an integer which reflects the contents of *obj* and the other Lisp objects it points to.

If two objects *obj1* and *obj2* are equal, then (**sxhash obj1**) and (**sxhash obj2**) are the same integer.

If the two objects are not equal, the values returned by **sxhash** are usually different, but not always; once in a rare while, by luck, you will encounter two distinct-looking objects that give the same result from **sxhash**.

This example creates a hash table whose keys are strings that are compared case-insensitively.

```
(defun case-fold-string= (a b)
  (compare-strings a nil nil b nil nil t))
(defun case-fold-string-hash (a)
  (sxhash (upcase a)))

(define-hash-table-test 'case-fold
  'case-fold-string= 'case-fold-string-hash)

(make-hash-table :test 'case-fold)
```

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)

(make-hash-table :test 'contents-hash)
```

## 7.4 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 7.1 [Creating Hash], page 97).

**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*.

## 8 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 11 [Variables], page 135, and Chapter 12 [Functions], page 160. For the precise read syntax for symbols, see Section 2.3.4 [Symbol Type], page 13.

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.

### 8.1 Symbol Components

Each symbol has four components (or “cells”), each of which references another object:

Print name

The *print name cell* holds a string that names the symbol for reading and printing. See `symbol-name` in Section 8.3 [Creating Symbols], page 104.

Value

The *value cell* holds the current value of the symbol as a variable. When a symbol is used as a form, the value of the form is the contents of the symbol’s value cell. See `symbol-value` in Section 11.7 [Accessing Variables], page 143.

Function

The *function cell* holds the function definition of the symbol. When a symbol is used as a function, its function definition is used in its place. This cell is also used to make a symbol stand for a keymap or a keyboard macro, for editor command execution. Because each symbol has separate value and function cells, variables names and function names do not conflict. See `symbol-function` in Section 12.8 [Function Cells], page 171.

Property list

The *property list cell* holds the property list of the symbol. See `symbol-plist` in Section 8.4 [Property Lists], page 107.

The print name cell always holds a string, and cannot be changed. The other three cells can be set individually to any specified Lisp object.

The print name cell holds the string that is the name of the 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. (In GNU Emacs Lisp, this lookup uses a hashing algorithm and an obarray; see Section 8.3 [Creating Symbols], page 104.)

The value cell holds the symbol’s value as a variable (see Chapter 11 [Variables], page 135). That is what you get if you evaluate the symbol as a Lisp expression (see Chapter 9 [Evaluation], page 110). Any Lisp object is a legitimate value. Certain 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 11.2 [Constant Variables], page 135.

We often refer to “the function `foo`” when we really mean the function stored in the function cell of the symbol `foo`. We make the distinction explicit only when necessary.

In normal usage, the function cell usually contains a function (see Chapter 12 [Functions], page 160) or a macro (see Chapter 13 [Macros], page 176), as that is what the Lisp interpreter expects to see there (see Chapter 9 [Evaluation], page 110). Keyboard macros (see Section 21.15 [Keyboard Macros], page 345), keymaps (see Chapter 22 [Keymaps], page 347) and autoload objects (see Section 9.1.8 [Autoloading], page 115) are also sometimes stored in the function cells of symbols.

The property list cell normally should hold a correctly formatted property list (see Section 8.4 [Property Lists], page 107), as a number of functions expect to see a property list there.

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’.

The four functions `symbol-name`, `symbol-value`, `symbol-plist`, and `symbol-function` return the contents of the four cells of a symbol. Here as an example we show the contents of the four cells of the symbol `buffer-file-name`:

```
(symbol-name 'buffer-file-name)
             ⇒ "buffer-file-name"
(symbol-value 'buffer-file-name)
             ⇒ "/gnu/elisp/symbols.texi"
(symbol-function 'buffer-file-name)
             ⇒ #<subr buffer-file-name>
(symbol-plist 'buffer-file-name)
             ⇒ (variable-documentation 29529)
```

Because this symbol is the variable which holds the name of the file being visited in the current buffer, the value cell contents we see are the name of the source file of this chapter of the Emacs Lisp Manual. The property list cell contains the list (`variable-documentation 29529`) which tells the documentation functions where to find the documentation string for the variable `buffer-file-name` in the ‘DOC-version’ file. (29529 is the offset from the beginning of the ‘DOC-version’ file to where that documentation string begins—see Section 24.1 [Documentation Basics], page 425.) The function cell contains the function for returning the name of the file. `buffer-file-name` names a primitive function, which has no read syntax and prints in hash notation (see Section 2.3.15 [Primitive Function Type], page 22). A symbol naming a function written in Lisp would have a lambda expression (or a byte-code object) in this cell.

## 8.2 Defining Symbols

A *definition* in Lisp is a special form that announces your intention to use a certain symbol in a particular way. In Emacs Lisp, you can define a symbol as a variable, or define it as a function (or macro), or both independently.

A definition construct 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. They are documented in detail in Section 11.5 [Defining Variables], page 139. For defining user option variables that can be customized, use `defcustom` (see Chapter 14 [Customization], page 185).

`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 12 [Functions], page 160.

`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 13 [Macros], page 176.

In Emacs Lisp, a definition is not required in order to use a symbol as a variable or function. Thus, you can make a symbol a global variable with `setq`, whether you define it first or not. The real purpose of definitions is to guide programmers and programming tools. They inform programmers who read the code that certain symbols are *intended* to be used as variables, or as functions. In addition, utilities such as ‘`etags`’ and ‘`make-docfile`’ recognize definitions, and add appropriate information to tag tables and the ‘`DOC-version`’ file. See Section 24.2 [Accessing Documentation], page 426.

## 8.3 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 set of characters. Failure to do so would cause complete confusion.

When the Lisp reader encounters a symbol, it reads all the characters of the name. Then it “hashes” those characters to find an index in a table called an *obarray*. Hashing is an efficient method of looking something up. 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 7 [Hash Tables], page 97.)

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 need 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.

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.

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.

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:** In Common Lisp, a single symbol may be interned in several obarrays.

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:** Changing the string by substituting characters does change the name of the symbol, but fails to update the obarray, so don't do it!

**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
```

**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 24.2 [Accessing Documentation], page 426, for another example using `mapatoms`.

**unintern symbol &optional 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**.

## 8.4 Property Lists

A *property list* (*plist* for short) is a list of paired elements stored in the property list cell of a symbol. Each of the pairs associates a property name (usually a symbol) with a property or value. Property lists are generally used to record information about a symbol, such as its documentation as a variable, the name of the file where it was defined, or perhaps even the grammatical class of the symbol (representing a word) in a language-understanding system.

Character positions in a string or buffer can also have property lists. See Section 32.19 [Text Properties], page 615.

The property names and values in a property list can be any Lisp objects, but the names are usually symbols. Property list functions compare the property names using **eq**. Here is an example of a property list, found on the symbol **progn** when the compiler is loaded:

```
(lisp-indent-function 0 byte-compile byte-compile-progn)
```

Here **lisp-indent-function** and **byte-compile** are property names, and the other two elements are the corresponding values.

### 8.4.1 Property Lists and Association Lists

Association lists (see Section 5.8 [Association Lists], page 81) 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 of its associations in one association list, it will typically need to search that entire list each time it checks for an association. This 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.

### 8.4.2 Property List Functions for Symbols

**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 36 [Abbrevs], page 699).

**get** *symbol property*

[Function]

This function finds the value of the property named *property* in *symbol*'s property list. If there is no such property, *nil* is returned. 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))
```

### 8.4.3 Property Lists Outside Symbols

These functions are useful for manipulating property lists that are stored in places other than symbols:

**plist-get** *plist property*

[Function]

This returns the value of the *property* property stored in the property list *plist*. For example,

```
(plist-get '(foo 4) 'foo)
          ⇒ 4
(plist-get '(foo 4 bad) 'foo)
          ⇒ 4
(plist-get '(foo 4 bad) 'bar)
          ⇒ wrong-type-argument error
```

It accepts a malformed *plist* argument and always returns `nil` if *property* is not found in the *plist*. For example,

```
(plist-get '(foo 4 bad) 'bar)
⇒ nil
```

**plist-put** *plist* *property* *value* [Function]

This stores *value* as the value of the *property* property in the property list *plist*. 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 '(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))
```

You could define `put` in terms of `plist-put` as follows:

```
(defun put (symbol prop value)
  (setplist symbol
            (plist-put (symbol-plist symbol) prop value)))
```

**lax-plist-get** *plist* *property* [Function]

Like `plist-get` except that it compares properties using `equal` instead of `eq`.

**lax-plist-put** *plist* *property* *value* [Function]

Like `plist-put` except that it compares properties using `equal` instead of `eq`.

**plist-member** *plist* *property* [Function]

This returns non-`nil` if *plist* contains the given *property*. 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*.

## 9 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 an *expression* or a *form*. The fact that expressions 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.

It is very common to read a Lisp expression and then evaluate the expression, 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` whether this object is a form to be evaluated, or serves some entirely different purpose. See Section 19.3 [Input Functions], page 271.

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 invoke the command. The execution of the command itself involves evaluation if the command is written in Lisp, but that is not a part of command key interpretation itself. See Chapter 21 [Command Loop], page 304.

Evaluation is a recursive process. That is, evaluation of a form may call `eval` to evaluate parts of the form. For example, evaluation of a function call first evaluates each argument of the function call, and then evaluates each form in the function body. Consider evaluation of the form `(car x)`: the subform `x` must first be evaluated recursively, so that its value can be passed as an argument to the function `car`.

Evaluation of a function call ultimately calls the function specified in it. See Chapter 12 [Functions], page 160. The execution of the function may itself work by evaluating the function definition; or the function may be a Lisp primitive implemented in C, or it may be a byte-compiled function (see Chapter 16 [Byte Compilation], page 214).

The evaluation of forms takes place in a context called the *environment*, which consists of the current values and bindings of all Lisp variables.<sup>1</sup> Whenever a form refers to a variable without creating a new binding for it, the value of the variable's binding in the current environment is used. See Chapter 11 [Variables], page 135.

Evaluation of a form may create new environments for recursive evaluation by binding variables (see Section 11.3 [Local Variables], page 136). These environments are temporary and vanish by the time evaluation of the form is complete. The form may also make changes that persist; these changes are called *side effects*. An example of a form that produces side effects is `(setq foo 1)`.

The details of what evaluation means for each kind of form are described below (see Section 9.1 [Forms], page 111).

---

<sup>1</sup> This definition of “environment” is specifically not intended to include all the data that can affect the result of a program.

## 9.1 Kinds of Forms

A Lisp object that is intended to be evaluated is called a *form*. 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 “all other types” which are self-evaluating forms.

### 9.1.1 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, evaluation of 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
```

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>
```

### 9.1.2 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 it has none (if its value cell is void), an error is signaled. For more information on the use of variables, see Chapter 11 [Variables], page 135.

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 11.2 [Constant Variables], page 135.

### 9.1.3 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.

### 9.1.4 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 12.3 [Function Names], page 165, for more information about using a symbol as a name for a function stored in the function cell of the symbol.

One possible consequence of this process is an infinite loop, in the event that a symbol’s function cell refers to the same symbol. Or a symbol may have a void function cell, in which case the subroutine `symbol-function` signals a `void-function` error. But if neither of these things happens, 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, the error `invalid-function` is signaled.

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 12.8 [Function Cells], page 171). 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`.

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 signals a `void-function` error if the final symbol is unbound and optional argument *noerror* is `nil` or omitted. Otherwise, if *noerror* is non-`nil`, it 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.

Here is how you could define `indirect-function` in Lisp:

```
(defun indirect-function (function)
  (if (symbolp function)
      (indirect-function (symbol-function function))
      function))
```

### 9.1.5 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 12.5 [Calling Functions], page 167). If the function is written in Lisp, the arguments are used to bind the argument variables of the function (see Section 12.2 [Lambda Expressions], page 161); 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.

### 9.1.6 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 13 [Macros], page 176, for a complete description of Emacs Lisp macros.

### 9.1.7 Special Forms

A *special form* is a primitive function 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.

Here is a list, in alphabetical order, of all of the special forms in Emacs Lisp with a reference to where each is described.

<code>and</code>	see Section 10.3 [Combining Conditions], page 122
<code>catch</code>	see Section 10.5.1 [Catch and Throw], page 125
<code>cond</code>	see Section 10.2 [Conditionals], page 120
<code>condition-case</code>	see Section 10.5.3.3 [Handling Errors], page 129
<code>defconst</code>	see Section 11.5 [Defining Variables], page 139
<code>defmacro</code>	see Section 13.4 [Defining Macros], page 178
<code>defun</code>	see Section 12.4 [Defining Functions], page 165
<code>defvar</code>	see Section 11.5 [Defining Variables], page 139
<code>function</code>	see Section 12.7 [Anonymous Functions], page 170
<code>if</code>	see Section 10.2 [Conditionals], page 120
<code>interactive</code>	see Section 21.3 [Interactive Call], page 310
<code>let</code>	
<code>let*</code>	see Section 11.3 [Local Variables], page 136
<code>or</code>	see Section 10.3 [Combining Conditions], page 122

**prog1**  
**prog2**  
**progn** see Section 10.1 [Sequencing], page 119  
**quote** see Section 9.2 [Quoting], page 115  
**save-current-buffer**  
    see Section 27.2 [Current Buffer], page 481  
**save-excursion**  
    see Section 30.3 [Excursions], page 568  
**save-restriction**  
    see Section 30.4 [Narrowing], page 569  
**save-window-excursion**  
    see Section 28.18 [Window Configurations], page 526  
**setq** see Section 11.8 [Setting Variables], page 143  
**setq-default**  
    see Section 11.10.2 [Creating Buffer-Local], page 149  
**track-mouse**  
    see Section 29.13 [Mouse Tracking], page 547  
**unwind-protect**  
    see Section 10.5 [Nonlocal Exits], page 125  
**while** see Section 10.4 [Iteration], page 124  
**with-output-to-temp-buffer**  
    see Section 38.8 [Temporary Displays], page 752

**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. `defun` is a special form in Emacs Lisp, but a macro in 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).

### 9.1.8 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. See Section 15.5 [Autoload], page 206.

## 9.2 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.

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
      ⇒ (quote foo)
'(quote foo)
      ⇒ (quote foo)
['foo]
      ⇒ [(quote foo)]
```

Other quoting constructs include **function** (see Section 12.7 [Anonymous Functions], page 170), which causes an anonymous lambda expression written in Lisp to be compiled, and ``'' (see Section 13.5 [Backquote], page 179), which is used to quote only part of a list, while computing and substituting other parts.

### 9.3 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.

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 15 [Loading], page 201).

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** [Function]

This is the basic function evaluating an expression. It evaluates *form* in the current environment and returns the result. How the evaluation proceeds depends on the type of the object (see Section 9.1 [Forms], page 111).

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 19.5 [Output Functions], page 273), as well as the values that result from evaluating the expressions in the region are printed using *stream*. See Section 19.4 [Output Streams], page 271.

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 202) 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*. *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 15.9 [Unloading], page 211), and defaults to `buffer-file-name` (see Section 27.4 [Buffer File Name], page 485). If *unibyte* is non-`nil`, `read` converts strings to unibyte whenever possible.

`eval-current-buffer` is an alias for this command.

**max-lisp-eval-depth** [Variable]

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 one way 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.

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 300. 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.

`max-specpdl-size` provides another limit on nesting. See [Local Variables], page 137.

**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` in `lisp-interaction-mode`.) The elements 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 is 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 Control Structures

A Lisp program consists of expressions or *forms* (see Section 9.1 [Forms], page 111). 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 13 [Macros], page 176).

### 10.1 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 control constructs likewise evaluate a series of forms but return a different value:

**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"
```

## 10.2 Conditionals

Conditional control structures choose among alternatives. Emacs Lisp has four conditional forms: **if**, which is much the same as in other languages; **when** and **unless**, which are variants of **if**; and **cond**, which is a generalized case statement.

**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 10.1 [Sequencing], page 119.)

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 the example below, `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 the value of the last of *body-forms* becomes the value of the `cond`. The 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*.

If every *condition* evaluates to `nil`, so that every clause fails, `cond` returns `nil`.

A clause may also look like this:

```
(condition)
```

Then, if *condition* is non-`nil` when tested, the value of *condition* becomes the value of the `cond` form.

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))
```

### 10.3 Constructs for Combining Conditions

This section describes three 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.

`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.

## 10.4 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 10.5 [Nonlocal Exits], page 125).

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. Here is an example of using **dotimes** to do something 100 times:

```
(dotimes (i 100)
  (insert "I will not obey absurd orders\n"))
```

## 10.5 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.

### 10.5.1 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 11.3 [Local Variables], page 136). Likewise, `throw` restores the buffer and position saved by `save-excursion` (see Section 30.3 [Excursions], page 568), and the narrowing status saved by `save-restriction` and the window selection saved by `save-window-excursion` (see Section 28.18 [Window Configurations], page 526). It also runs any cleanups established with the `unwind-protect` special form when it exits that form (see Section 10.5.4 [Cleanups], page 133).

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 21.12 [Recursive Editing], page 342).

**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`.

`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`).

### 10.5.2 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 “`go to`.”) 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.

### 10.5.3 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 10.5.4 [Cleanups], page 133.) 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.

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 10.5.1 [Catch and Throw], page 125.

### 10.5.3.1 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 10.5.3.2 [Processing of Errors], page 129). 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 21.10 [Quitting], page 338.

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` (see Section 4.7 [Formatting Strings], page 56) 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 "You have committed %d errors" 10)
  error You have committed 10 errors
```

`error` works by calling `signal` with two arguments: the error symbol `error`, and a list containing the string returned by `format`.

**Warning:** If you want to use your own string as an error message verbatim, don’t just write `(error string)`. If `string` contains ‘%’, it will be interpreted as a format specifier, with undesirable results. Instead, use `(error "%s" string)`.

`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 bearing a property `error-conditions` whose value is a list of condition names. This is how Emacs Lisp classifies different sorts of errors. See Section 10.5.3.4 [Error Symbols], page 132, 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-arg` 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 10.5.3.3 [Handling Errors], page 129).

The function `signal` never returns (though in older Emacs versions it could sometimes return).

```
(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"
```

**Common Lisp note:** Emacs Lisp has nothing like the Common Lisp concept of continuable errors.

### 10.5.3.2 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. The debugger is enabled if the variable `debug-on-error` (see Section 18.1.1 [Error Debugging], page 237) 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.

### 10.5.3.3 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.

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. See Section 18.1.1 [Error Debugging], page 237. 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.

When an error is handled, control returns to the handler. Before this happens, Emacs unbinds all variable bindings made by binding constructs that are being exited and executes the cleanups of all `unwind-protect` forms that are exited. Once control arrives at the handler, the body of the handler is executed.

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.

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.

Error signaling and handling have some resemblance to `throw` and `catch` (see Section 10.5.1 [Catch and Throw], page 125), 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 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; *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. The `error-conditions` property of this symbol is a list of condition names (see Section 10.5.3.4 [Error Symbols], page 132). 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.

#### `error-message-string` *error-description* [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 128.

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, at least not 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 signaled with `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
```

#### 10.5.3.4 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.

In order for a symbol to be an error symbol, it must have an `error-conditions` property which gives a list of condition names. This list defines the conditions that this kind of error belongs to. (The error symbol itself, and the symbol `error`, should always be members of this list.) Thus, the hierarchy of condition names is defined by the `error-conditions` properties of the error symbols. Because quitting is not considered an error, the value of the `error-conditions` property of `quit` is just (`quit`).

In addition to the `error-conditions` list, the error symbol should have an `error-message` property whose value is a string to be printed when that error is signaled but not handled. If the error symbol has no `error-message` property or if the `error-message` property exists, but is not a string, the error message ‘peculiar error’ is used. See [Definition of signal], page 128.

Here is how we define a new error symbol, `new-error`:

```
(put 'new-error
      'error-conditions
      '(error my-own-errors new-error))
⇒ (error my-own-errors new-error)
(put 'new-error 'error-message "A new error")
⇒ "A new error"
```

This error has three condition names: `new-error`, the narrowest classification; `my-own-errors`, which we imagine is a wider classification; and `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 128) 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 the three 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 891, for a list of all the standard error symbols and their conditions.

#### 10.5.4 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 32.25 [Atomic Changes], page 637.)

`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.

The number of currently active `unwind-protect` forms counts, together with the number of local variable bindings, against the limit `max-specpdl-size` (see [Local Variables], page 137).

For example, here we make an invisible buffer for temporary use, and make sure to kill it before finishing:

```
(save-excursion
  (let ((buffer (get-buffer-create " *temp*")))
    (set-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 another `save-excursion` 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 483). 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 37 [Processes], page 705) 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.

# 11 Variables

A *variable* is a name used in a program to stand for a value. Nearly all programming languages have variables of some sort. In the text of a Lisp program, variables are written using the syntax for symbols.

In Lisp, unlike most programming languages, programs are represented primarily as Lisp objects and only secondarily as text. The Lisp objects used for variables are symbols: the symbol name is the variable name, and the variable's value is stored in the value cell of the symbol. The use of a symbol as a variable is independent of its use as a function name. See Section 8.1 [Symbol Components], page 102.

The Lisp objects that constitute a Lisp program determine the textual form of the program—it is simply the read syntax for those Lisp objects. This is why, for example, a variable in a textual Lisp program is written using the read syntax for the symbol that represents the variable.

## 11.1 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` 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 by 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
```

## 11.2 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 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.

```

### 11.3 Local Variables

Global variables have values that last until explicitly superseded with new values. Sometimes it is useful to create variable values that exist temporarily—only until a certain part of the program finishes. These values are called *local*, and the variables so used are called *local variables*.

For example, when a function is called, its argument variables receive new local values that last until the function exits. The `let` special form explicitly establishes new local values for specified variables; these last until exit from the `let` form.

Establishing a local value saves away the previous value (or lack of one) of the variable. When the life span of the local value is over, the previous value is restored. In the meantime, we say that the previous value is *shadowed* and *not visible*. Both global and local values may be shadowed (see Section 11.9.1 [Scope], page 145).

If you set a variable (such as with `setq`) while it is local, this replaces the local value; it does not alter the global value, or previous local values, that are shadowed. To model this behavior, we speak of a *local binding* of the variable as well as a local value.

The local binding is a conceptual place that holds a local value. Entry to a function, or a special form such as `let`, creates the local binding; exit from the function or from the `let` removes the local binding. As long as the local binding lasts, the variable's value is stored within it. Use of `setq` or `set` while there is a local binding stores a different value into the local binding; it does not create a new binding.

We also speak of the *global binding*, which is where (conceptually) the global value is kept.

A variable can have more than one local binding at a time (for example, if there are nested `let` forms that bind it). In such a case, the most recently created local binding that still exists is the *current binding* of the variable. (This rule is called *dynamic scoping*; see Section 11.9 [Variable Scoping], page 145.) If there are no local bindings, the variable's global binding is its current binding. We sometimes call the current binding the *most-local existing binding*, for emphasis. Ordinary evaluation of a symbol always returns the value of its current binding.

The special forms `let` and `let*` exist to create local bindings.

`let (bindings...) forms...` [Special Form]

This special form binds variables according to *bindings* and then evaluates all of the *forms* in textual order. The `let`-form returns the value of the last form in *forms*.

Each of the *bindings* is either (i) a symbol, in which case that symbol is bound to `nil`; or (ii) a list of the form `(symbol value-form)`, in which case *symbol* is bound to the result of evaluating *value-form*. If *value-form* is omitted, `nil` is used.

All of the *value-forms* 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)
```

### **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 reasonably 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)
```

Here is a complete list of the other facilities that create local bindings:

- Function calls (see Chapter 12 [Functions], page 160).
- Macro calls (see Chapter 13 [Macros], page 176).
- **condition-case** (see Section 10.5.3 [Errors], page 127).

Variables can also have buffer-local bindings (see Section 11.10 [Buffer-Local Variables], page 147) and frame-local bindings (see Section 11.11 [Frame-Local Variables], page 153); a few variables have terminal-local bindings (see Section 29.2 [Multiple Displays], page 530). These kinds of bindings work somewhat like ordinary local bindings, but they are localized depending on “where” you are in Emacs, rather than localized in time.

### **max-specpdl-size** [Variable]

This variable defines the limit on the total number of local variable bindings and **unwind-protect** cleanups (see Section 10.5.4 [Cleaning Up from Nonlocal Exits], page 133) that are allowed before signaling an error (with data “**Variable binding depth exceeds max-specpdl-size**”).

This limit, with the associated error when it is exceeded, is one way that Lisp avoids infinite recursion on an ill-defined function. **max-lisp-eval-depth** provides another limit on depth of nesting. See [Eval], page 117.

The default value is 1000. Entry to the Lisp debugger increases the value, if there is little room left, to make sure the debugger itself has room to execute.

## 11.4 When a Variable is “Void”

If you have never given a symbol any value as a global variable, we say that that symbol’s global value is *void*. In other words, the symbol’s value cell does not have any Lisp object in it. If you try to evaluate the symbol, you get a **void-variable** error rather than a value.

Note that a value of **nil** is not the same as void. 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 *a value*. A void variable does not have any value.

After you have given a variable a value, you can make it void once more using **makunbound**.

### **makunbound** *symbol*

[Function]

This function makes the current variable binding of *symbol* void. Subsequent attempts to use this symbol’s value as a variable will signal the error **void-variable**, unless and until you set it again.

**makunbound** returns *symbol*.

```
(makunbound 'x)           ; Make the global value of x void.
⇒ x
x
[error] Symbol's value as variable is void: x
```

If *symbol* is locally bound, **makunbound** affects the most local existing binding. This is the only way a symbol can have a void local binding, since all the constructs that create local bindings create them with values. In this case, the voidness lasts at most as long as the binding does; when the binding is removed due to exit from the construct that made it, the previous local or global binding is reexposed as usual, and the variable is no longer void unless the newly reexposed binding was void all along.

```
(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
```

A variable that has been made void with **makunbound** is indistinguishable from one that has never received a value and has always been void.

You can use the function **boundp** to test whether a variable is currently void.

**boundp variable** [Function]

`boundp` returns `t` if `variable` (a symbol) is not void; more precisely, if its current binding is not void. It returns `nil` otherwise.

```
(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
```

## 11.5 Defining Global Variables

You may announce your intention to use a symbol as a global variable with a *variable definition*: a special form, either `defconst` or `defvar`.

In Emacs Lisp, definitions serve three purposes. First, they inform people who read the code that certain symbols are *intended* to be used a certain way (as variables). Second, they inform the Lisp system of these things, supplying a value and documentation. Third, they provide information to utilities such as `etags` and `make-docfile`, which create data bases of the functions and variables in a program.

The difference between `defconst` and `defvar` is primarily a matter of intent, serving to inform human readers of whether the value should ever change. Emacs Lisp does not restrict the ways in which a variable can be used based on `defconst` or `defvar` declarations. However, it does make a difference for initialization: `defconst` unconditionally initializes the variable, while `defvar` initializes it only if it is void.

**defvar symbol [value [doc-string]]** [Special Form]

This special form defines `symbol` as a variable and can also initialize and document it.

The definition informs a person reading your code that `symbol` is used as a variable that might be set or changed. Note that `symbol` is not evaluated; the symbol to be defined must appear explicitly in the `defvar`.

If `symbol` is void and `value` is specified, `defvar` evaluates it and sets `symbol` to the result. But if `symbol` already has a value (i.e., it is not void), `value` is not even evaluated, and `symbol`'s value remains unchanged. If `value` is omitted, the value of `symbol` is not changed in any case.

If `symbol` has a buffer-local binding in the current buffer, `defvar` operates on the default value, which is buffer-independent, not the current (buffer-local) binding. It sets the default value if the default value is void. See Section 11.10 [Buffer-Local Variables], page 147.

When you evaluate a top-level `defvar` 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.

If the `doc-string` argument appears, it specifies the documentation for the variable. (This opportunity to specify documentation is one of the main benefits of defining

the variable.) The documentation is stored in the symbol's `variable-documentation` property. The Emacs help functions (see Chapter 24 [Documentation], page 425) look for this property.

If the variable is a user option that users would want to set interactively, you should use '\*' as the first character of `doc-string`. This lets users set the variable conveniently using the `set-variable` command. Note that you should nearly always use `defcustom` instead of `defvar` to define these variables, so that users can use `M-x customize` and related commands to set them. See Chapter 14 [Customization], page 185.

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 following form changes the documentation string for `bar`, making it a user option, but does not change the value, since `bar` already has a value. (The addition `(1+ nil)` would get an error if it were evaluated, but since it is not evaluated, there is no error.)

```
(defvar bar (1+ nil)
  "*The normal weight of a bar.")
  ⇒ bar
bar
  ⇒ 23
```

Here is an equivalent expression for the `defvar` special form:

```
(defvar symbol value doc-string)
≡
(progn
  (if (not (boundp 'symbol))
    (setq symbol value))
  (if 'doc-string
    (put 'symbol 'variable-documentation 'doc-string))
  'symbol)
```

The `defvar` form returns `symbol`, but it is normally used at top level in a file where its value does not matter.

**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`.

`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`.)

Here, `pi` is a constant that presumably ought not to be changed by anyone (attempts by the Indiana State Legislature notwithstanding). As the second form illustrates, however, this is only advisory.

```
(defconst pi 3.1415 "Pi to five places.")
  ⇒ pi
(setq pi 3)
  ⇒ pi
pi
  ⇒ 3
```

`user-variable-p variable` [Function]

This function returns `t` if `variable` is a user option—a variable intended to be set by the user for customization—and `nil` otherwise. (Variables other than user options exist for the internal purposes of Lisp programs, and users need not know about them.)

User option variables are distinguished from other variables either though being declared using `defcustom`<sup>1</sup> or by the first character of their `variable-documentation` property. If the property exists and is a string, and its first character is ‘\*’, then the variable is a user option. Aliases of user options are also user options.

If a user option variable has a `variable-interactive` property, the `set-variable` command uses that value to control reading the new value for the variable. The property’s value is used as if it were specified in `interactive` (see Section 21.2.1 [Using Interactive], page 305). However, this feature is largely obsoleted by `defcustom` (see Chapter 14 [Customization], page 185).

**Warning:** If the `defconst` and `defvar` special forms are used while the variable has a local binding (made with `let`, or a function argument), they set the local-binding’s value; the top-level binding is not changed. This is not what you usually want. To prevent it, 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.

## 11.6 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 23.1 [Hooks], page 382).

‘`...-function`’

The value is a function.

‘`...-functions`’

The value is a list of functions.

‘`...-form`’

The value is a form (an expression).

---

<sup>1</sup> They may also be declared equivalently in ‘`cus-start.el`’.

**‘...-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 “good” arguments and `nil` for “bad” arguments.

**‘...-flag’**

The value is significant only as to whether it is `nil` or not.

**‘...-program’**

The value is a program name.

**‘...-command’**

The value is a whole shell command.

**‘...-switches’**

The value specifies options for a command.

When you define a variable, always consider whether you should mark it as “risky”; see Section 11.13 [File Local Variables], page 155.

When defining and initializing a variable that holds a complicated value (such as a keymap with bindings in it), it’s best to put the entire computation of the value into the `defvar`, like this:

```
(defvar my-mode-map
  (let ((map (make-sparse-keymap)))
    (define-key map "\C-c\C-a" 'my-command)
    ...
  map)
  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 run hooks to alter part of the contents (such as, to rebind keys). Third, evaluating the `defvar` form with `C-M-x` will reinitialize the map completely.

Putting so much code in the `defvar` form has one disadvantage: it puts the documentation string far away from the line which names the variable. Here’s a safe way to avoid that:

```
(defvar my-mode-map nil
  docstring)
(unless my-mode-map
  (let ((map (make-sparse-keymap)))
    (define-key map "\C-c\C-a" 'my-command)
    ...
  (setq my-mode-map map)))
```

This has all the same advantages as putting the initialization inside the `defvar`, except that you must type `C-M-x` twice, once on each form, if you do want to reinitialize the variable.

But be careful not to write the code like this:

```
(defvar my-mode-map nil
  docstring)
(unless my-mode-map
  (setq my-mode-map (make-sparse-keymap))
  (define-key my-mode-map "\C-c\C-a" 'my-command)
  ...)
```

This code sets the variable, then alters it, but it does so in more than one step. If the user quits just after the `setq`, that leaves the variable neither correctly initialized nor void nor `nil`. Once that happens, reloading the file will not initialize the variable; it will remain incomplete.

## 11.7 Accessing Variable Values

The usual way to reference a variable is to write the symbol which names it (see Section 9.1.2 [Symbol Forms], page 111). This requires you to specify the variable name when you write the program. Usually that is exactly what you want to do. Occasionally you need to choose at run time which variable to reference; then you can use `symbol-value`.

`symbol-value symbol` [Function]

This function returns the value of *symbol*. This is the value in the innermost local binding of the symbol, or its global value if it has no local bindings.

```
(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
```

A `void-variable` error is signaled if the current binding of *symbol* is void.

## 11.8 How to Alter a Variable Value

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 most-local existing 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 sets *symbol*'s value to *value*, then returns *value*. Since **set** is a function, the expression written for *symbol* is evaluated to obtain the symbol to set.

The most-local existing binding of the variable is the binding that is set; shadowed bindings are not affected.

```
(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)
```

Logically speaking, `set` is a more fundamental primitive than `setq`. Any use of `setq` can be trivially rewritten to use `set`; `setq` could even be defined as a macro, given the availability of `set`. However, `set` itself is rarely used; beginners hardly need to know about it. It is useful only for choosing at run time which variable to set. For example, the command `set-variable`, which reads a variable name from the user and then sets the variable, needs to use `set`.

**Common Lisp note:** In Common Lisp, `set` always changes the symbol's "special" or dynamic value, ignoring any lexical bindings. In Emacs Lisp, all variables and all bindings are dynamic, so `set` always affects the most local existing binding.

## 11.9 Scoping Rules for Variable Bindings

A given symbol `foo` can have several local variable bindings, established at different places in the Lisp program, as well as a global binding. The most recently established binding takes precedence over the others.

Local bindings in Emacs Lisp have *indefinite scope* and *dynamic extent*. *Scope* refers to *where* textually in the source code the binding can be accessed. "Indefinite scope" means that any part of the program can potentially access the variable binding. *Extent* refers to *when*, as the program is executing, the binding exists. "Dynamic extent" means that the binding lasts as long as the activation of the construct that established it.

The combination of dynamic extent and indefinite scope is called *dynamic scoping*. By contrast, most programming languages use *lexical scoping*, in which references to a local variable must be located textually within the function or block that binds the variable.

**Common Lisp note:** Variables declared "special" in Common Lisp are dynamically scoped, like all variables in Emacs Lisp.

### 11.9.1 Scope

Emacs Lisp uses *indefinite scope* for local variable bindings. This means that any function anywhere in the program text might access a given binding of a variable. Consider the following function definitions:

```
(defun binder (x)      ; x is bound in binder.
  (foo 5))            ; foo is some other function.

(defun user ()        ; x is used "free" in user.
  (list x))
```

In a lexically scoped language, the binding of `x` in `binder` would never be accessible in `user`, because `user` is not textually contained within the function `binder`. However, in dynamically-scoped Emacs Lisp, `user` may or may not refer to the binding of `x` established in `binder`, depending on the circumstances:

- If we call `user` directly without calling `binder` at all, then whatever binding of `x` is found, it cannot come from `binder`.
- If we define `foo` as follows and then call `binder`, then the binding made in `binder` will be seen in `user`:

```
(defun foo (lose)
  (user))
```

- However, if we define `foo` as follows and then call `binder`, then the binding made in `binder` *will not* be seen in `user`:

```
(defun foo (x)
  (user))
```

Here, when `foo` is called by `binder`, it binds `x`. (The binding in `foo` is said to *shadow* the one made in `binder`.) Therefore, `user` will access the `x` bound by `foo` instead of the one bound by `binder`.

Emacs Lisp uses dynamic scoping because simple implementations of lexical scoping are slow. In addition, every Lisp system needs to offer dynamic scoping at least as an option; if lexical scoping is the norm, there must be a way to specify dynamic scoping instead for a particular variable. It might not be a bad thing for Emacs to offer both, but implementing it with dynamic scoping only was much easier.

### 11.9.2 Extent

*Extent* refers to the time during program execution that a variable name is valid. In Emacs Lisp, a variable is valid only while the form that bound it is executing. This is called *dynamic extent*. “Local” or “automatic” variables in most languages, including C and Pascal, have dynamic extent.

One alternative to dynamic extent is *indefinite extent*. This means that a variable binding can live on past the exit from the form that made the binding. Common Lisp and Scheme, for example, support this, but Emacs Lisp does not.

To illustrate this, the function below, `make-add`, returns a function that purports to add `n` to its own argument `m`. This would work in Common Lisp, but it does not do the job in Emacs Lisp, because after the call to `make-add` exits, the variable `n` is no longer bound to the actual argument 2.

```
(defun make-add (n)
  (function (lambda (m) (+ n m))) ; Return a function.
  ⇒ make-add
  (fset 'add2 (make-add 2)) ; Define function add2
  ; with (make-add 2).
  ⇒ (lambda (m) (+ n m))
  (add2 4) ; Try to add 2 to 4.
  [error] Symbol's value as variable is void: n
```

Some Lisp dialects have “closures,” objects that are like functions but record additional variable bindings. Emacs Lisp does not have closures.

### 11.9.3 Implementation of Dynamic Scoping

A simple sample implementation (which is not how Emacs Lisp actually works) may help you understand dynamic binding. This technique is called *deep binding* and was used in early Lisp systems.

Suppose there is a stack of bindings, which are variable-value pairs. At entry to a function or to a `let` form, we can push bindings onto the stack for the arguments or local

variables created there. We can pop those bindings from the stack at exit from the binding construct.

We can find the value of a variable by searching the stack from top to bottom for a binding for that variable; the value from that binding is the value of the variable. To set the variable, we search for the current binding, then store the new value into that binding.

As you can see, a function’s bindings remain in effect as long as it continues execution, even during its calls to other functions. That is why we say the extent of the binding is dynamic. And any other function can refer to the bindings, if it uses the same variables while the bindings are in effect. That is why we say the scope is indefinite.

The actual implementation of variable scoping in GNU Emacs Lisp uses a technique called *shallow binding*. Each variable has a standard place in which its current value is always found—the value cell of the symbol.

In shallow binding, setting the variable works by storing a value in the value cell. Creating a new binding works by pushing the old value (belonging to a previous binding) onto a stack, and storing the new local value in the value cell. Eliminating a binding works by popping the old value off the stack, into the value cell.

We use shallow binding because it has the same results as deep binding, but runs faster, since there is never a need to search for a binding.

#### 11.9.4 Proper Use of Dynamic Scoping

Binding a variable in one function and using it in another is a powerful technique, but if used without restraint, it can make programs hard to understand. There are two clean ways to use this technique:

- Use or bind the variable only in a few related functions, written close together in one file. Such a variable is used for communication within one program.

You should write comments to inform other programmers that they can see all uses of the variable before them, and to advise them not to add uses elsewhere.

- Give the variable a well-defined, documented meaning, and make all appropriate functions refer to it (but not bind it or set it) wherever that meaning is relevant. For example, the variable `case-fold-search` is defined as “non-`nil` means ignore case when searching”; various search and replace functions refer to it directly or through their subroutines, but do not bind or set it.

Then you can bind the variable in other programs, knowing reliably what the effect will be.

In either case, you should define the variable with `defvar`. This helps other people understand your program by telling them to look for inter-function usage. It also avoids a warning from the byte compiler. Choose the variable’s name to avoid name conflicts—don’t use short names like `x`.

### 11.10 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: *buffer-local* bindings, which apply only in one buffer, and *frame-local* bindings, which apply

only in one frame. Having different values for a variable in different buffers and/or frames is an important customization method.

This section describes buffer-local bindings; for frame-local bindings, see the following section, Section 11.11 [Frame-Local Variables], page 153. (A few variables have bindings that are local to each terminal; see Section 29.2 [Multiple Displays], page 530.)

### 11.10.1 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 (assuming there are no frame-local bindings to complicate the matter), 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 23.2 [Major Modes], page 384.

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 or frame-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 or frame-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*.

### 11.10.2 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.
(save-excursion
  (set-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.

If the variable is terminal-local, this function signals an error. Such variables cannot have buffer-local bindings as well. See Section 29.2 [Multiple Displays], page 530.

**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`.

**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.

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`.

**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.

**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` will become buffer-local in buffer `buffer` (which defaults to the current buffer) if it is set there.

**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 11.10.3 [Default Value], page 152) of `variable` instead.

**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.) It returns an association list (see Section 5.8 [Association Lists], page 81) in which each element contains one buffer-local variable and its value. However, when a variable's buffer-local binding in `buffer` is void, then the variable appears directly in the resulting list.

```
(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**

[Function]

This function eliminates all the buffer-local variable bindings of the current buffer except for variables marked as “permanent.” As a result, the buffer will see the default values of most variables.

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 23.1 [Hooks], page 382.

A buffer-local variable is *permanent* if the variable name (a symbol) has a `permanent-local` property that is non-`nil`. 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.

### 11.10.3 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 or frame-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 11.7 [Accessing Variables], page 143).

**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

```

## 11.11 Frame-Local Variables

Just as variables can have buffer-local bindings, they can also have frame-local bindings. These bindings belong to one frame, and are in effect when that frame is selected. Frame-local bindings are actually frame parameters: you create a frame-local binding in a specific frame by calling `modify-frame-parameters` and specifying the variable name as the parameter name.

To enable frame-local bindings for a certain variable, call the function `make-variable-frame-local`.

```
make-variable-frame-local variable [Command]
Enable the use of frame-local bindings for variable. This does not in itself create any frame-local bindings for the variable; however, if some frame already has a value for variable as a frame parameter, that value automatically becomes a frame-local binding.
```

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.

If the variable is terminal-local, this function signals an error, because such variables cannot have frame-local bindings as well. See Section 29.2 [Multiple Displays], page 530. A few variables that are implemented specially in Emacs can be buffer-local, but can never be frame-local.

This command returns *variable*.

Buffer-local bindings take precedence over frame-local bindings. Thus, consider a variable `foo`: if the current buffer has a buffer-local binding for `foo`, that binding is active; otherwise, if the selected frame has a frame-local binding for `foo`, that binding is active; otherwise, the default binding of `foo` is active.

Here is an example. First we prepare a few bindings for `foo`:

```
(setq f1 (selected-frame))
(make-variable-frame-local 'foo)

;; Make a buffer-local binding for foo in 'b1'.
(set-buffer (get-buffer-create "b1"))
(make-local-variable 'foo)
(setq foo '(b 1))

;; Make a frame-local binding for foo in a new frame.
;; Store that frame in f2.
(setq f2 (make-frame))
(modify-frame-parameters f2 '((foo . (f 2))))
```

Now we examine `foo` in various contexts. Whenever the buffer ‘`b1`’ is current, its buffer-local binding is in effect, regardless of the selected frame:

```
(select-frame f1)
(set-buffer (get-buffer-create "b1"))
foo
⇒ (b 1)

(select-frame f2)
(set-buffer (get-buffer-create "b1"))
foo
⇒ (b 1)
```

Otherwise, the frame gets a chance to provide the binding; when frame `f2` is selected, its frame-local binding is in effect:

```
(select-frame f2)
(set-buffer (get-buffer "*scratch*"))
foo
⇒ (f 2)
```

When neither the current buffer nor the selected frame provides a binding, the default binding is used:

```
(select-frame f1)
(select-frame f1)
(set-buffer (get-buffer "*scratch*"))
foo
⇒ nil
```

When the active binding of a variable is a frame-local binding, setting the variable changes that binding. You can observe the result with `frame-parameters`:

```
(select-frame f2)
(select-frame f2)
(set-buffer (get-buffer "*scratch*"))
(setq foo 'nobody)
(assq 'foo (frame-parameters f2))
⇒ (foo . nobody)
```

## 11.12 Possible Future Local Variables

We have considered the idea of bindings that are local to a category of frames—for example, all color frames, or all frames with dark backgrounds. We have not implemented them because it is not clear that this feature is really useful. You can get more or less the same results by adding a function to `after-make-frame-functions`, set up to define a particular frame parameter according to the appropriate conditions for each frame.

It would also be possible to implement window-local bindings. We don't know of many situations where they would be useful, and it seems that indirect buffers (see Section 27.11 [Indirect Buffers], page 494) with buffer-local bindings offer a way to handle these situations more robustly.

If sufficient application is found for either of these two kinds of local bindings, we will provide it in a subsequent Emacs version.

## 11.13 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 processing of file local variables.

**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.

**hack-local-variables &optional mode-only** [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 23.2.3 [Auto Major Mode], page 388).

If the optional argument `mode-only` is non-`nil`, then all this function does is return `t` if the ‘`-*`’ line or the local variables list specifies a mode and `nil` otherwise. It does not set the mode nor any other file local variable.

If a file local variable could specify a function that would be called later, or an expression that would be executed later, simply visiting a file could take over your Emacs. Emacs takes several measures to prevent this.

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 standardly have `safe-local-variable` properties, including `fill-column`, `fill-prefix`, and `indent-tabs-mode`. For boolean-valued variables that are safe, use `booleanp` as the property value. Lambda expressions should be quoted so that `describe-variable` can display the predicate.

#### `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.

#### `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*. A 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 considered risky. The variables ‘`font-lock-keywords`’, ‘`font-lock-keywords`’ followed by a digit, and ‘`font-lock-syntactic-keywords`’ are also considered risky. Finally, any variable whose name has a non-`nil` `risky-local-variable` property is considered risky.

#### `risky-local-variable-p sym`

[Function]

This function returns non-`nil` if `sym` is a risky variable, based on the above criteria.

If a variable is risky, it will not be entered automatically into `safe-local-variable-values` as described above. Therefore, Emacs will always query before setting a risky variable, unless the user explicitly allows the setting by customizing `safe-local-variable-values` directly.

#### `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.

## 11.14 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-alia base-variable &optional docstring**

[Function]

This function defines the symbol `new-alia` as a variable alias for symbol `base-variable`. This means that retrieving the value of `new-alia` returns the value of `base-variable`, and changing the value of `new-alia` 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-alia`; otherwise, the alias gets the same documentation as `base-variable` has, if any, unless `base-variable` is itself an alias, in which case `new-alia` 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 &optional when**

[Function]

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.

If provided, `when` should be a string indicating when the variable was first made obsolete—for example, a date or a release number.

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` [Macro]  
`&optional when docstring`

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)
```

`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
```

## 11.15 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 879, 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 only take on integer values. Attempting to assign them any other value will result in an error:

```
(setq window-min-height 5.0)
[error] Wrong type argument: integerp, 5.0
```

## 12 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.

### 12.1 What Is a Function?

In a general sense, a function is a rule for carrying on a computation given several values called *arguments*. The result of the computation is called the value of the function. The computation can also have side effects: lasting changes in the values of variables or the contents of data structures.

Here are important terms for functions in Emacs Lisp and for other function-like objects.

*function* In Emacs Lisp, a *function* is anything that can be applied to arguments in a Lisp program. In some cases, we use it more specifically to mean a function written in Lisp. Special forms and macros are not functions.

*primitive* A *primitive* is a function callable from Lisp that is written in C, such as `car` or `append`. These functions are also called *built-in functions*, or *subrs*. (Special forms are also considered primitives.)

Usually the reason we implement a function as a primitive is either because it is fundamental, because it provides a low-level interface to operating system services, or because it needs to run fast. Primitives can be modified or added only by changing the C sources and recompiling the editor. See Section E.5 [Writing Emacs Primitives], page 876.

*lambda expression*

A *lambda expression* is a function written in Lisp. These are described in the following section.

*special form*

A *special form* is 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. Many special forms are described in Chapter 10 [Control Structures], page 119.

*macro* A *macro* is a construct defined in Lisp by the programmer. It differs from a function in that it translates a Lisp expression that you write into an equivalent expression 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 13 [Macros], page 176, for how to define and use macros.

*command* A *command* is an object that `command-execute` can invoke; it is a possible definition for a key sequence. Some functions are commands; a function written in Lisp is a command if it contains an interactive declaration (see Section 21.2 [Defining Commands], page 305). Such a function can be called from Lisp expressions like other functions; in this case, the fact that the function is a command makes no difference.

Keyboard macros (strings and vectors) are commands also, even though they are not functions. A symbol is a command if its function definition is a com-

mand; such symbols can be invoked with *M-x*. The symbol is a function as well if the definition is a function. See Section 21.3 [Interactive Call], page 310.

#### **keystroke command**

A *keystroke command* is a command that is bound to a key sequence (typically one to three keystrokes). The distinction is made here merely to avoid confusion with the meaning of “command” in non-Emacs editors; for Lisp programs, the distinction is normally unimportant.

#### **byte-code function**

A *byte-code function* is a function that has been compiled by the byte compiler. See Section 2.3.16 [Byte-Code Type], page 22.

#### **functionp object**

[Function]

This function returns *t* if *object* is any kind of function, or a special form, or, recursively, a symbol whose function definition is a function or special form. (This does not include macros.)

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
```

#### **subr-arity subr**

[Function]

This function provides information about the argument list of a primitive, *subr*. The returned value is a pair (*min* . *max*). *min* is the minimum number of args. *max* is the maximum number or the symbol `many`, for a function with `&rest` arguments, or the symbol `unevalled` if *subr* is a special form.

## 12.2 Lambda Expressions

A function written in Lisp is a list that looks like this:

```
(lambda (arg-variables...)  
  [documentation-string]  
  [interactive-declaration]  
  body-forms...)
```

Such a list is called a *lambda expression*. In Emacs Lisp, it actually is valid as an expression—it evaluates to itself. In some other Lisp dialects, a lambda expression is not a valid expression at all. In either case, its main use is not to be evaluated as an expression, but to be called as a function.

### 12.2.1 Components of a Lambda Expression

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. 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 11.3 [Local Variables], page 136.

The documentation string is a Lisp string object placed within the function definition to describe the function for the Emacs help facilities. See Section 12.2.4 [Function Documentation], page 164.

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 21.2 [Defining Commands], page 305, 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.

### 12.2.2 A Simple Lambda-Expression Example

Consider for example the following function:

```
(lambda (a b c) (+ a b c))
```

We can call this function by writing it as the CAR of an expression, like this:

```
((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:

```
((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.

It is not often useful to write a lambda expression as the CAR of a form in this way. You can get the same result, of making local variables and giving them values, using the special form `let` (see Section 11.3 [Local Variables], page 136). And `let` is clearer and easier to use. In practice, lambda expressions are either stored as the function definitions of symbols, to produce named functions, or passed as arguments to other functions (see Section 12.7 [Anonymous Functions], page 170).

However, calls to explicit lambda expressions were very useful in the old days of Lisp, before the special form `let` was invented. At that time, they were the only way to bind and initialize local variables.

### 12.2.3 Other 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.

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. If there are only two arguments, `c` is `nil`; if two or three arguments, `d` is `nil`; if four arguments or fewer, `e` is `nil`.

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:

```

((lambda (n) (1+ n))           ; One required:
 1)                           ; requires exactly one argument.
   ⇒ 2
((lambda (n &optional n1)       ; One required and one optional:
   (if n1 (+ n n1) (1+ n)))  ; 1 or 2 arguments.
 1 2)
   ⇒ 3
((lambda (n &rest ns)          ; One required and one rest:
   (+ n (apply '+ ns)))       ; 1 or more arguments.
 1 2 3 4 5)
   ⇒ 15

```

#### 12.2.4 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 24 [Documentation], page 425, 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.

## 12.3 Naming a Function

In most computer languages, every function has a name; the idea of a function without a name is nonsensical. In Lisp, a function in the strictest sense has no name. It is simply a list whose first element is `lambda`, a byte-code function object, or a primitive subr-object.

However, a symbol can serve as the name of a function. This happens when you put the function in the symbol's *function cell* (see Section 8.1 [Symbol Components], page 102). Then the symbol itself becomes a valid, callable function, equivalent to the list or subr-object that its function cell refers to. 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 function indirection*; see Section 9.1.4 [Function Indirection], page 112.

In practice, nearly all functions are given names in this way and referred to through their names. For example, the symbol `car` works as a function and does what it does because the primitive subr-object `#<subr car>` is stored in its function cell.

We give functions names because it is convenient to refer to them by their names in Lisp expressions. For primitive subr-objects such as `#<subr car>`, names are the only way you can refer to them: there is no read syntax for such objects. For functions written in Lisp, the name is more convenient to use in a call than an explicit lambda expression. Also, a function with a name can refer to itself—it can be recursive. Writing the function's name in its own definition is much more convenient than making the function definition point to itself (something that is not impossible but that has various disadvantages in practice).

We often identify functions with the symbols used to name them. For example, we often speak of “the function `car`,” not distinguishing between the symbol `car` and the primitive subr-object that is its function definition. For most purposes, the distinction is not important.

Even so, keep in mind that a function need not have a unique name. While a given function object *usually* appears in the function cell of only one symbol, this is just a matter of convenience. It is easy to store it in several symbols using `fset`; then each of the symbols is equally well a name for the same function.

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. (Some Lisp dialects, such as Scheme, do not distinguish between a symbol's value and its function definition; a symbol's value as a variable is also its function definition.) If you have not given a symbol a function definition, you cannot use it as a function; whether the symbol has a value as a variable makes no difference to this.

## 12.4 Defining Functions

We usually give a name to a function when it is first created. This is called *defining a function*, and it is done with the `defun` special form.

`defun name argument-list body-forms` [Special Form]

`defun` is the usual way to define new Lisp functions. It defines the symbol `name` as a function that looks like this:

`(lambda argument-list . body-forms)`

`defun` stores this lambda expression in the function cell of *name*. It returns the value *name*, but usually we ignore this value.

As described previously, *argument-list* is a list of argument names and may include the keywords `&optional` and `&rest` (see Section 12.2 [Lambda Expressions], page 161). Also, the first two of the *body-forms* may be a documentation string and an interactive declaration.

There is no conflict if the same symbol *name* is also used as a variable, since the symbol's value cell is independent of the function cell. See Section 8.1 [Symbol Components], page 102.

Here are some examples:

```
(defun foo () 5)
  ⇒ foo
(foo)
  ⇒ 5

(defun bar (a &optional b &rest c)
  (list a b c))
  ⇒ bar
(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 ()
  "Uppcase the last letter of a word."
  (interactive)
  (backward-word 1)
  (forward-word 1)
  (backward-char 1)
  (capitalize-word 1))
  ⇒ capitalize-backwards
```

Be careful not to redefine existing functions unintentionally. `defun` redefines even primitive functions such as `car` without any hesitation or notification. Redefining a function already defined is often done deliberately, and there is no way to distinguish deliberate redefinition from unintentional redefinition.

**defalias** *name definition &optional docstring* [Function]

This special form defines the symbol *name* as a function, with definition *definition* (which can be any valid Lisp function). It returns *definition*.

If *docstring* is non-`nil`, it becomes the function documentation of *name*. Otherwise, any documentation provided by *definition* is used.

The proper place to use `defalias` is where a specific function 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 15.9 [Unloading], page 211).

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 12.8 [Function Cells], page 171.

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 open-code it. See Section 12.10 [Inline Functions], page 173.

## 12.5 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 9 [Evaluation], page 110, 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.

```
(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` 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)
```

For an interesting example of using `apply`, see [Definition of `mapcar`], page 169.

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 two different kinds of no-op function:

**identity** *arg* [Function]

This function returns *arg* and has no side effects.

**ignore** *&rest args* [Function]

This function ignores any arguments and returns `nil`.

## 12.6 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; `mapcar` and `mapconcat`, which scan a list, are described here. See [Definition of `mapatoms`], page 106, for the function `mapatoms` which maps over the symbols in an obarray. See [Definition of `maphash`], page 99, 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 93).

**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 'char-to-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))
```

**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 separator* [Function]

**mapconcat** applies *function* to each element of *sequence*: the results, which must be strings, are concatenated. Between each pair of result strings, **mapconcat** inserts the string *separator*. Usually *separator* contains a space or comma or other suitable punctuation.

The argument *function* must be a function that can take one argument and return a string. 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 (function (lambda (x) (format "%c" (1+ x))))
  "HAL-8000"
  "")
⇒ "IBM.9111"
```

## 12.7 Anonymous Functions

In Lisp, a function is a list that starts with `lambda`, a byte-code function compiled from such a list, or alternatively a primitive subr-object; names are “extra.” Although usually functions are defined with `defun` and given names at the same time, it is occasionally more concise to use an explicit lambda expression—an anonymous function. Such a list is valid wherever a function name is.

Any method of creating such a list makes a valid function. Even this:

```
(setq silly (append '(lambda (x)) (list (list '+ (* 3 4) 'x))))
⇒ (lambda (x) (+ 12 x))
```

This computes a list that looks like `(lambda (x) (+ 12 x))` and makes it the value (*not* the function definition!) of `silly`.

Here is how we might call this function:

```
(funcall silly 1)
⇒ 13
```

(It does *not* work to write `(silly 1)`, because this function is not the *function definition* of `silly`. We have not given `silly` any function definition, just a value as a variable.)

Most of the time, anonymous functions are constants that appear in your program. For example, you might want to pass one as an argument to the function `mapcar`, which applies any given function to each element of a list.

Here we define a function `change-property` which uses a function as its third argument:

```
(defun change-property (symbol prop function)
  (let ((value (get symbol prop)))
    (put symbol prop (funcall function value))))
```

Here we define a function that uses `change-property`, passing it a function to double a number:

```
(defun double-property (symbol prop)
  (change-property symbol prop '(lambda (x) (* 2 x))))
```

In such cases, we usually use the special form `function` instead of simple quotation to quote the anonymous function, like this:

```
(defun double-property (symbol prop)
  (change-property symbol prop
    (function (lambda (x) (* 2 x)))))
```

Using `function` instead of `quote` makes a difference if you compile the function `double-property`. For example, if you compile the second definition of `double-property`, the anonymous function is compiled as well. By contrast, if you compile the first definition which uses ordinary `quote`, the argument passed to `change-property` is the precise list shown:

```
(lambda (x) (* x 2))
```

The Lisp compiler cannot assume this list is a function, even though it looks like one, since it does not know what `change-property` will do with the list. Perhaps it will check whether the CAR of the third element is the symbol `*`! Using `function` tells the compiler it is safe to go ahead and compile the constant function.

Nowadays it is possible to omit `function` entirely, like this:

```
(defun double-property (symbol prop)
  (change-property symbol prop (lambda (x) (* 2 x))))
```

This is because `lambda` itself implies `function`.

We sometimes write `function` instead of `quote` when quoting the name of a function, but this usage is just a sort of comment:

```
(function symbol) ≡ (quote symbol) ≡ 'symbol
```

The read syntax `#'` is a short-hand for using `function`. For example,

```
#'(lambda (x) (* x x))
```

is equivalent to

```
(function (lambda (x) (* x x)))
```

#### `function` *function-object*

[Special Form]

This special form returns *function-object* without evaluating it. In this, it is equivalent to `quote`. However, it serves as a note to the Emacs Lisp compiler that *function-object* is intended to be used only as a function, and therefore can safely be compiled. Contrast this with `quote`, in Section 9.2 [Quoting], page 115.

See [describe-symbols example], page 427, for a realistic example using `function` and an anonymous function.

## 12.8 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 113.

#### `symbol-function` *symbol*

[Function]

This returns the object in the function cell of *symbol*. If the symbol's function cell is `void`, a `void-function` error is signaled.

This function does not check that the returned object is a legitimate function.

```
(defun bar (n) (+ n 2))
  ⇒ bar
(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 such a symbol as a function, it 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 they can be valid functions if you define them in turn 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 11.4 [Void Variables], page 138.)

```
(defun foo (x) x)
      ⇒ foo
(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.

There are three normal uses of this function:

- Copying one symbol's function definition to another—in other words, making an alternate name for a function. (If you think of this as the definition of the new name, you should use `defalias` instead of `fset`; see [Definition of defalias], page 166.)
- Giving a symbol a function definition that is not a list and therefore cannot be made with `defun`. For example, you can use `fset` to give a symbol `s1` a function definition which is another symbol `s2`; then `s1` serves as an alias for whatever definition `s2` presently has. (Once again use `defalias` instead of `fset` if you think of this as the definition of `s1`.)
- In constructs for defining or altering functions. If `defun` were not a primitive, it could be written in Lisp (as a macro) using `fset`.

Here are examples of these uses:

```
;; Save foo's definition in old-foo.
(fset 'old-foo (symbol-function 'foo))
```

```
;; Make the symbol car the function definition of xfirst.
;; (Most likely, defalias would be better than fset here.)
(fset 'xfirst 'car)
      ⇒ car
(xfirst '(1 2 3))
      ⇒ 1
```

```
(symbol-function 'xfirst)
  ⇒ car
(symbol-function (symbol-function 'xfirst))
  ⇒ #<subr car>

;; Define a named keyboard macro.
(fset 'kill-two-lines "\^u2\^k")
  ⇒ "\^u2\^k"

;; Here is a function that alters other functions.
(defun copy-function-definition (new old)
  "Define NEW with the same function definition as OLD."
  (fset new (symbol-function old)))
```

`fset` is sometimes used to save the old definition of a function before redefining it. That permits the new definition to invoke the old definition. But it is unmodular and unclean for a Lisp file to redefine a function defined elsewhere. If you want to modify a function defined by another package, it is cleaner to use `defadvice` (see Chapter 17 [Advising Functions], page 226).

## 12.9 Declaring Functions Obsolete

You can use `make-obsolete` to declare a function obsolete. This indicates that the function may be removed at some stage in the future.

`make-obsolete obsolete-name current-name &optional when` [Function]

This function makes the byte compiler warn that the function `obsolete-name` is obsolete. 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. If `current-name` is a string, it is the warning message.

If provided, `when` should be a string indicating when the function was first made obsolete—for example, a date or a release number.

You can define a function as an alias and declare it obsolete at the same time using the macro `define-obsolete-function-alias`.

`define-obsolete-function-alias obsolete-name current-name &optional when docstring` [Macro]

This 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 docstring)
(make-obsolete obsolete-name current-name when))
```

## 12.10 Inline Functions

You can define an *inline function* by using `defsubst` instead of `defun`. An inline function works just like an ordinary function except for one thing: when you compile a call to the function, the function's definition is open-coded into the caller.

Making a function inline makes explicit 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 Chapter 17 [Advising Functions], page 226). 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.

It’s possible to define a macro to expand into the same code that an inline function would execute. (See Chapter 13 [Macros], page 176.) 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 very easy; simply 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. (See Section 13.6.2 [Argument Evaluation], page 180.)

Inline functions can be used and open-coded later on in the same file, following the definition, just like macros.

## 12.11 Determining whether a Function is Safe to Call

Some major modes such as SES call functions that are stored in user files. (See Info file ‘`ses`’, node ‘Top’, 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.

## 12.12 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 12.5 [Calling Functions], page 167.

**autoload** See Section 15.5 [Autoload], page 206.

**call-interactively**  
See Section 21.3 [Interactive Call], page 310.

**commandp** See Section 21.3 [Interactive Call], page 310.

**documentation**  
See Section 24.2 [Accessing Documentation], page 426.

**eval** See Section 9.3 [Eval], page 116.

**funcall** See Section 12.5 [Calling Functions], page 167.

**function** See Section 12.7 [Anonymous Functions], page 170.

**ignore** See Section 12.5 [Calling Functions], page 167.

**indirect-function**  
See Section 9.1.4 [Function Indirection], page 112.

**interactive**  
See Section 21.2.1 [Using Interactive], page 305.

**interactive-p**  
See Section 21.3 [Interactive Call], page 310.

**mapatoms** See Section 8.3 [Creating Symbols], page 104.

**mapcar** See Section 12.6 [Mapping Functions], page 168.

**map-char-table**  
See Section 6.6 [Char-Tables], page 93.

**mapconcat**  
See Section 12.6 [Mapping Functions], page 168.

**undefined**  
See Section 22.11 [Functions for Key Lookup], page 360.

## 13 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 12.10 [Inline Functions], page 173.

### 13.1 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`.

### 13.2 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 not the value of the macro call. Instead, it is an alternate expression for computing that value, 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.

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)))
⇒ inc

(macroexpand '(inc r))
⇒ (setq r (1+ r))

(defmacro inc2 (var1 var2)
  (list 'progn (list 'inc var1) (list 'inc var2)))
⇒ inc2

(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)))
```

### 13.3 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. To make this feature work, you must put the `defmacro` in the same file where it is used, and before its first use.

Byte-compiling a file executes any `require` calls at top-level in the file. This is in case the file needs the required packages for proper compilation. One way to ensure that necessary macro definitions are available during compilation is to require the files that define them (see Section 15.7 [Named Features], page 209). To avoid loading the macro definition files when someone *runs* the compiled program, write `eval-when-compile` around the `require` calls (see Section 16.5 [Eval During Compile], page 219).

## 13.4 Defining Macros

A Lisp macro is a list whose CAR is `macro`. Its CDR should be a function; expansion of the macro works by applying the function (with `apply`) to the list of unevaluated argument-expressions 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 usually defined with the special form `defmacro`.

`defmacro name argument-list body-forms...` [Special Form]

`defmacro` defines the symbol `name` as a macro that looks like this:

```
(macro lambda argument-list . body-forms)
```

(Note that the CDR of this list is a function—a lambda expression.) This macro object is stored in the function cell of `name`. The value returned by evaluating the `defmacro` form is `name`, but usually we ignore this value.

The shape and meaning of `argument-list` is the same as in a function, and the keywords `&rest` and `&optional` may be used (see Section 12.2.3 [Argument List], page 163). Macros may have a documentation string, but any `interactive` declaration is ignored since macros cannot be called interactively.

The body of the macro definition can include a `declare` form, which can specify how TAB should indent macro calls, and how to step through them for Edebug.

`declare specs...` [Macro]

A `declare` form is used in a macro definition to specify various additional information about it. Two kinds of specification are currently supported:

```
(debug edebug-form-spec)
```

Specify how to step through macro calls for Edebug. See Section 18.2.15.1 [Instrumenting Macro Calls], page 258.

```
(indent indent-spec)
```

Specify how to indent calls to this macro. See Section 13.7 [Indenting Macros], page 184, for more details.

A `declare` form only has its special effect in the body of a `defmacro` form if it immediately follows the documentation string, if present, or the argument list otherwise. (Strictly speaking, *several* `declare` forms can follow the documentation string or argument list, but since a `declare` form can have several *specs*, they can always be combined into a single form.) When used at other places in a `defmacro` form, or outside a `defmacro` form, `declare` just returns `nil` without evaluating any *specs*.

No macro absolutely needs a `declare` form, because that form has no effect on how the macro expands, on what the macro means in the program. It only affects secondary features: indentation and Edebug.

## 13.5 Backquote

Macros often need to construct large list structures from a mixture of constants and non-constant parts. To make this easier, use the “`” syntax (usually called *backquote*).

Backquote allows you to quote a list, but selectively evaluate elements of that list. In the simplest case, it is identical to the special form `quote` (see Section 9.2 [Quoting], page 115). 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. Backquote evaluates the argument of ‘,’ and puts the value in the list structure:

```
(list 'a 'list 'of (+ 2 3) 'elements)
  ⇒ (a list of 5 elements)
'(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:

```
(defmacro t-becomes-nil (variable)
  '(if (eq ,variable t)
       (setq ,variable nil)))

(t-becomes-nil foo)
  ≡ (if (eq foo t) (setq foo nil))
```

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)
```

In old Emacs versions, before version 19.29, “`” used a different syntax which required an extra level of parentheses around the entire backquote construct. Likewise, each ‘,’ or ‘,@’ substitution required an extra level of parentheses surrounding both the ‘,’ or ‘,@’ and the following expression. The old syntax required whitespace between the “`”, ‘,’ or ‘,@’ and the following expression.

This syntax is still accepted, for compatibility with old Emacs versions, but we recommend not using it in new programs.

## 13.6 Common Problems Using Macros

The basic facts of macro expansion have counterintuitive consequences. This section describes some important consequences that can lead to trouble, and rules to follow to avoid trouble.

### 13.6.1 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.

### 13.6.2 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 simple “for” loop such as one might find in Pascal.

```
(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

(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.

### 13.6.3 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 8.3 [Creating Symbols], page 104). 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.

### 13.6.4 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 9.3 [Eval], page 116). If the argument is supposed to refer to the user's variables, you may have trouble if the user happens to use a variable with the same name as one of the macro arguments. Inside the macro body, the macro argument binding is the most local binding of this 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))
  ⇒ foo
(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`.

Another problem with calling `eval` in a macro definition is that it probably won't do what you intend in a compiled program. The byte-compiler runs macro definitions while compiling the program, when the program's own computations (which you might have wished to access with `eval`) don't occur and its local variable bindings don't exist.

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.

### 13.6.5 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.

## 13.7 Indenting Macros

You can use the `declare` form in the macro definition to specify how to TAB should indent indent calls to the macro. You write it like this:

```
(declare (indent indent-spec))
```

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:

- `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.
- `pos` The position at which the line being indented begins.

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.

# 14 Writing Customization Definitions

This chapter describes how to declare user options for customization, and also customization groups for classifying them. We use the term *customization item* to include both kinds of customization definitions—as well as face definitions (see Section 38.12.1 [Defining Faces], page 763).

## 14.1 Common Item Keywords

All kinds of customization declarations (for variables and groups, and for faces) accept keyword arguments for specifying various information. This section describes some keywords that apply to all kinds.

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.** One legitimate case for use of `:tag` is to specify a dash where normally a hyphen would be converted to a space:

```
(defcustom cursor-in-non-selected-windows ...
  :tag "Cursor In Non-selected Windows")
```

### `:group group`

Put this customization item in group *group*. 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 an active field which 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.
- (**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.
- (**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’.

An item can have more than one external link; however, most items have none at all.

- :load** *file*
  - Load file *file* (a string) before displaying this customization item. Loading is done with **load-library**, 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.
- :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. The value of *package* is a symbol and *version* is a string.

This keyword takes priority over `:version`.

`package` should be the official name of the package, such as MH-E or Gnus. 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 look like this:

```
(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.

## 14.2 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 twelve 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 14.1 [Common Keywords], page 185), you can also use this keyword in `defgroup`:

`:prefix prefix`

If the name of an item in the group starts with `prefix`, then the tag for that item is constructed (by default) by omitting `prefix`.

One group can have any number of prefixes.

The prefix-dropping feature is currently turned off, which means that `:prefix` currently has no effect. We did this because we found that dropping the specified prefixes often led to confusing names for options. This happened because the people who wrote the `defgroup` definitions for various groups added `:prefix` keywords whenever they make logical sense—that is, whenever the variables in the library have a common prefix.

In order to obtain good results with `:prefix`, it would be necessary to check the specific effects of dropping a particular prefix, given the specific items in a group and their names and documentation. If the resulting text is not clear, then `:prefix` should not be used in that case.

It should be possible to recheck all the customization groups, delete the `:prefix` specifications which give unclear results, and then turn this feature back on, if someone would like to do the work.

### 14.3 Defining Customization Variables

Use `defcustom` to declare user-customizable variables.

`defcustom option standard doc [keyword value]... [Macro]`

This construct declares `option` as a customizable user option variable. You should not quote `option`. The argument `doc` specifies the documentation string for the variable. There is no need to start it with a ‘\*’, because `defcustom` automatically marks `option` as a user option (see Section 11.5 [Defining Variables], page 139).

The argument `standard` is an expression that specifies the standard value for `option`. Evaluating the `defcustom` form evaluates `standard`, but does not necessarily install the standard value. If `option` already has a default value, `defcustom` does not change it. If the user has saved a customization for `option`, `defcustom` installs the user’s customized value as `option`’s default value. If neither of those cases applies, `defcustom` installs the result of evaluating `standard` as the default value.

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. We recommend avoiding backquotes in `standard`, because they are not expanded when editing the value, so list values will appear to have the wrong structure.

Every `defcustom` should specify `:group` at least once.

If you specify the `:set` keyword, to make the variable take other special actions when set through the customization buffer, the variable’s documentation string should tell the user specifically how to do the same job in hand-written Lisp code.

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 11.5 [Defining Variables], page 139.

`defcustom` accepts the following additional 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 14.4 [Customization Types], page 191, for more information.

**`: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`.

**`:set setfunction`**

Specify *setfunction* as the way to change the value of this option. 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). The default for *setfunction* is `set-default`.

**`: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-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` 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`.

**custom-initialize-safe-set****custom-initialize-safe-default**

These functions behave like `custom-initialize-set` (`custom-initialize-default`, respectively), but catch errors. If an error occurs during initialization, they set the variable to `nil` using `set-default`, and throw no error.

These two functions are only meant for options defined in pre-loaded files, where some variables or functions used to compute the option's value may not yet be defined. The option normally gets updated in '`startup.el`', ignoring the previously computed value. Because of this typical usage, the value which these two functions compute normally only matters when, after startup, one unsets the option's value and then reevaluates the defcustom. By that time, the necessary variables and functions will be defined, so there will not be an error.

**:set-after variables**

When setting variables according to saved customizations, make sure to set the variables `variables` before this one; in other words, 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.

The `:require` keyword is useful for an option that turns on the operation of a certain feature. Assuming that the package is coded to check the value of the option, you still need to arrange for the package to be loaded. You can do that with `:require`. See Section 14.1 [Common Keywords], page 185. Here is an example, from the library '`saveplace.el`':

```
(defcustom save-place nil
  "Non-nil means automatically save place in each file..."
  :type 'boolean
  :require 'saveplace)
```

```
:group 'save-place)
```

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`.

Internally, `defcustom` uses the symbol property `standard-value` to record the expression for the standard value, and `saved-value` to record the value saved by the user with the customization buffer. Both properties are actually lists whose car is an expression which evaluates to the value.

## 14.4 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 14.4.4 [Type Keywords], page 198).

Some of the 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.

### 14.4.1 Simple Types

This section describes all the simple customization types.

`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.

<b>integer</b>	The value must be an integer, and is represented textually in the customization buffer.
<b>number</b>	The value must be a number (floating point or integer), and is represented textually in the customization buffer.
<b>float</b>	The value must be a floating point number, and is represented textually in the customization buffer.
<b>string</b>	The value must be a string, and the customization buffer shows just the contents, with no delimiting ‘"’ characters and no quoting with ‘\’.
<b>regexp</b>	Like <b>string</b> except that the string must be a valid regular expression.
<b>character</b>	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.
<b>file</b>	The value must be a file name, and you can do completion with <i>M-TAB</i> .
<b>(file :must-match t)</b>	The value must be a file name for an existing file, and you can do completion with <i>M-TAB</i> .
<b>directory</b>	The value must be a directory name, and you can do completion with <i>M-TAB</i> .
<b>hook</b>	The value must be a list of functions (or a single function, but that is obsolete usage). This customization type is used for hook variables. You can use the <b>:options</b> keyword in a hook variable’s <b>defcustom</b> to specify a list of functions recommended for use in the hook; see Section 14.3 [Variable Definitions], page 188.
<b>alist</b>	The value must be a list of cons-cells, the CAR of each cell representing a key, and the CDR of the same cell representing an associated value. The user can add and delete key/value pairs, and edit both the key and the value of each pair.  You can specify the key and value types like this:  <code>(alist :key-type key-type :value-type value-type)</code>
	where <b>key-type</b> and <b>value-type</b> are customization type specifications. The default key type is <b>sexp</b> , and the default value type is <b>sexp</b> .  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 <b>:options</b> (see Section 14.3 [Variable Definitions], page 188). 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 <b>:options</b> keyword argument.  The argument to the <b>:options</b> keywords should be a list of specifications for reasonable keys in the alist. Ordinarily, they are simply atoms, which stand for themselves as. 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)))

(defcustom pets '(("brian")
                 ("dorith" "dog" "guppy")
                 ("ken" "cat"))
  "Alist of people's pets.
  In an element (KEY . VALUE), KEY is the person's name,
  and the VALUE is a list of that person's pets."
  :type '(alist :value-type (repeat string)))
```

### plist

The `plist` custom type is similar to the `alist` (see above), except that the information is stored as a property list, i.e. a list of this form:

```
(key value key value key value ...)
```

The default :key-type for plist is symbol, rather than sexp.

<b>symbol</b>	The value must be a symbol. It appears in the customization buffer as the name of the symbol.
<b>function</b>	The value must be either a lambda expression or a function name. When it is a function name, you can do completion with <i>M-TAB</i> .
<b>variable</b>	The value must be a variable name, and you can do completion with <i>M-TAB</i> .
<b>face</b>	The value must be a symbol which is a face name, and you can do completion with <i>M-TAB</i> .
<b>boolean</b>	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.
<b>coding-system</b>	The value must be a coding-system name, and you can do completion with <i>M-TAB</i> .
<b>color</b>	The value must be a valid color name, and you can do completion with <i>M-TAB</i> . A sample is provided.

### 14.4.2 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 the CDR are displayed and edited separately, each according to the type that you specify for it.

**(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.

`(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`.

`(choice alternative-types...)`

The value must fit at least 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 14.4.4 [Type Keywords], page 198.

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 documentation 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 has no side effects, and 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:

<code>:tag tag</code>	Use <code>tag</code> as the name of this alternative, for user communication purposes. This is useful for a type that appears inside of a <code>choice</code> .
<code>:match-alternatives criteria</code>	Use <code>criteria</code> to match possible values. This is used only in <code>restricted-sexp</code> .
<code>:args argument-list</code>	Use the elements of <code>argument-list</code> as the arguments of the type construct. For instance, <code>(const :args (foo))</code> is equivalent to <code>(const foo)</code> . You rarely need to write <code>:args</code> explicitly, because normally the arguments are recognized automatically as whatever follows the last keyword-value pair.

#### 14.4.3 Splicing into Lists

The `:inline` feature lets you splice a variable number of elements into the middle of a list or vector. You use it in a `set`, `choice` or `repeat` type which appears among the element-types of a list or vector.

Normally, each of the element-types in a list or vector describes one and only one element of the list or vector. Thus, if an element-type is a `repeat`, that specifies a list of unspecified length which appears as one element.

But when the element-type uses `:inline`, the value it matches is merged directly into the containing sequence. For example, if it matches a list with three elements, those become three elements of the overall sequence. This is analogous to using `'@` in the backquote construct.

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.

#### 14.4.4 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`

This is used 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`.

If `nil` is not a valid value for the alternative, then it is essential to specify a valid default with `:value`.

##### `: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 an active field 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.

#### 14.4.5 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 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.

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.

A file containing Lisp code is often called a *library*. Thus, the “Rmail library” is a file containing code for Rmail mode. Similarly, a “Lisp library directory” is a directory of files containing Lisp code.

### 15.1 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 15.5 [Autoload], page 206). `require` loads a file if it isn’t already loaded (see Section 15.7 [Named Features], page 209). 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, it is loaded. If there is no file by that name, then `load` looks for a file named ‘*filename.el*’. If that file exists, it is loaded. 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 perverse file names such as ‘*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), unless it contains an explicit directory name.

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 15.3 [Library Search], page 203.

If you get a warning that ‘foo.elc’ is older than ‘foo.el’, it means you should consider recompiling ‘foo.el’. See Chapter 16 [Byte Compilation], page 214.

When loading a source file (not compiled), `load` performs character set translation just as Emacs would do when visiting the file. See Section 33.10 [Coding Systems], page 648.

Messages like ‘Loading foo...’ and ‘Loading foo...done’ appear in the echo area during loading unless *nomessage* is non-*nil*.

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 the error `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 in how it reads its argument interactively.

#### `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-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.

Normally, the variable’s value is *nil*, which means those functions should use `read`.

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 117.

For information about how `load` is used in building Emacs, see Section E.1 [Building Emacs], page 870.

## 15.2 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.

## 15.3 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`

[User Option]

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 name) or `nil` (which stands for the current working directory).

The value of `load-path` is initialized from the environment variable `EMACSLOADPATH`, if that exists; otherwise its default value is specified in ‘`emacs/src/epaths.h`’ when Emacs is built. Then the list is expanded by adding subdirectories of the directories in the list.

The syntax of `EMACSLOADPATH` is the same as used for `PATH`; ‘`:`’ (or ‘`;`’, according to the operating system) separates directory names, and ‘`.`’ is used for the current default directory. Here is an example of how to set your `EMACSLOADPATH` variable from a `csh` ‘`.login`’ file:

```
setenv EMACSLOADPATH .:/user/bil/emacs:/usr/local/share/emacs/20.3/lisp
```

Here is how to set it using `sh`:

```
export EMACSLOADPATH
EMACSLOADPATH=.::/user/bil/emacs:/usr/local/share/emacs/20.3/lisp
```

Here is an example of code you can place in your init file (see Section 39.1.2 [Init File], page 813) to add several directories to the front of your default `load-path`:

```
(setq load-path
      (append (list nil "/user/bil/emacs"
                    "/usr/local/lisplib"
                    "~/emacs")
              load-path))
```

In this example, the path searches the current working directory first, followed then by the ‘`/user/bil/emacs`’ directory, the ‘`/usr/local/lisplib`’ directory, and the ‘`~/emacs`’ directory, which are then followed by the standard directories for Lisp code.

Dumping Emacs uses a special value of `load-path`. If the value of `load-path` at the end of dumping is unchanged (that is, still the same special value), the dumped Emacs switches to the ordinary `load-path` value when it starts up, as described above. But if `load-path` has any other value at the end of dumping, that value is used for execution of the dumped Emacs also.

Therefore, if you want to change `load-path` temporarily for loading a few libraries in ‘`site-init.el`’ or ‘`site-load.el`’, you should bind `load-path` locally with `let` around the calls to `load`.

The default value of `load-path`, when running an Emacs which has been installed on the system, includes two special directories (and their subdirectories as well):

```
"/usr/local/share/emacs/version/site-lisp"
```

and

```
"/usr/local/share/emacs/site-lisp"
```

The first one is for locally installed packages for a particular Emacs version; the second is for locally installed packages meant for use with all installed Emacs versions.

There are several reasons why a Lisp package that works well in one Emacs version can cause trouble in another. Sometimes packages need updating for incompatible changes in Emacs; sometimes they depend on undocumented internal Emacs data that can change without notice; sometimes a newer Emacs version incorporates a version of the package, and should be used only with that version.

Emacs finds these directories’ subdirectories and adds them to `load-path` when it starts up. Both immediate subdirectories and subdirectories multiple levels down are added to `load-path`.

Not all subdirectories are included, though. Subdirectories whose names do not start with a letter or digit are excluded. Subdirectories named ‘RCS’ or ‘CVS’ are excluded. Also, a subdirectory which contains a file named ‘.nosearch’ is excluded. You can use these methods to prevent certain subdirectories of the ‘site-lisp’ directories from being searched.

If you run Emacs from the directory where it was built—that is, an executable that has not been formally installed—then `load-path` normally contains two additional directories. These are the `lisp` and `site-lisp` subdirectories of the main build directory. (Both are represented as absolute file names.)

**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.

## 15.4 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 33.1 [Text Representations], page 640). 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 33.10 [Coding Systems], page 648.

To make the results more predictable, Emacs always performs decoding into the multibyte representation when loading Lisp files, even if it was started with the ‘--unibyte’ option. This means that string constants with non-ASCII characters translate into multibyte strings. The only exception is when a particular file specifies no decoding.

The reason Emacs is designed this way is so that Lisp programs give predictable results, regardless of how Emacs was started. In addition, this enables programs that depend on using multibyte text to work even in a unibyte Emacs. Of course, such programs should be designed to notice whether the user prefers unibyte or multibyte text, by checking `default-enable-multibyte-characters`, and convert representations appropriately.

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 ‘`--unibyte: t;--`’ in a comment on the file’s first line. With that designator, the file will unconditionally be interpreted as unibyte, even in an ordinary multibyte Emacs session. This can matter when making keybindings to non-ASCII characters written as `?vliteral`.

## 15.5 Autoload

The `autoload` facility allows you to make a function or macro known in Lisp, but put off loading the file that defines it. The first call to the function automatically reads the proper file to install the real definition and other associated code, then runs the real definition as if it had been loaded all along.

There are two ways to set up an autoloaded function: by calling `autoload`, and by writing a special “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 `update-file-autoloads`, 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`, then `autoload` 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.

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 `define-key`; 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, `autoload` does nothing and returns `nil`. If the function cell of *function* is void, or is already an autoload object, then it is defined as an autoload object like this:

```
(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-version’ file (see Section 24.1 [Documentation Basics], page 425), t means the function is interactive, and nil that it is not a macro or a keymap.

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 command *M-x update-file-autoloads* writes a corresponding `autoload` call into ‘`loaddefs.el`’. Building Emacs loads ‘`loaddefs.el`’ and thus calls `autoload`. *M-x update-directory-autoloads* is even more powerful; it updates autoloads for all files in the current directory.

The same magic comment can copy any kind of form into ‘`loaddefs.el`’. If the form following the magic comment is not a function-defining form or a `defcustom` form, it is copied verbatim. “Function-defining forms” include `define-skeleton`, `define-derived-mode`, `define-generic-mode` and `define-minor-mode` as well as `defun` and `defmacro`. To save space, a `defcustom` form is converted to a `defvar` in ‘`loaddefs.el`’, with some additional information if it uses `:require`.

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 *M-x update-file-autoloads* 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 (quote doctor) "doctor" "
Switch to *doctor* buffer and start giving psychotherapy.

\\(fn)" t nil)
```

The backslash and newline immediately following the double-quote are a convention used only in the preloaded uncompiled Lisp files such as ‘`loaddefs.el`’; they tell `make-docfile` to put the documentation string in the ‘`etc/DOC`’ file. See Section E.1 [Building Emacs], page 870. See also the commentary in ‘`lib-src/make-docfile.c`’. ‘`(fn)`’ in the usage part of the documentation string is replaced with the function’s name when the various help functions (see Section 24.5 [Help Functions], page 431) 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
...)
```

## 15.6 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 11.5 [Defining Variables], page 139.)

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, write this:

```
(or (assq 'leif-mode minor-mode-alist)
         (push '(leif-mode " Leif") minor-mode-alist))
```

or this:

```
(add-to-list '(leif-mode " Leif") minor-mode-alist)
```

Occasionally you will want to test explicitly whether a library has already been loaded. Here’s one way to test, in a library, whether it has been loaded before:

```
(defvar foo-was-loaded nil)

(unless foo-was-loaded
  execute-first-time-only
  (setq foo-was-loaded t))
```

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.

## 15.7 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 ‘`emacs/lisp/prolog.el`’, the definition for `run-prolog` includes the following code:

```
(defun run-prolog ()
  "Run an inferior Prolog process, with I/O via buffer *prolog*."
  (interactive)
  (require 'comint)
  (switch-to-buffer (make-comint "prolog" prolog-program-name))
  (inferior-prolog-mode))
```

The expression `(require 'comint)` loads the file ‘`comint.el`’ if it has not yet been loaded. This ensures that `make-comint` is defined. Features are normally named after the files that provide them, so that `require` need not be given the file name.

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 16 [Byte Compilation], page 214) 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 the list **features** if it is not already in the list. 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 37.16.3 [Network Feature Testing], page 730, 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 15.5 [Autoload], page 206.

**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, ‘Required feature *feature* was not provided’.

**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.

## 15.8 Which File Defined a Certain Symbol

**symbol-file** *symbol* &**optional** *type* [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** or **defvar**, that specifies function definition only or variable definition only.

The value is normally an absolute file name. It can also be **nil**, if the definition is not associated with any file.

The basis for **symbol-file** is the data in the variable **load-history**.

**load-history** [Variable]

This variable's value is an alist connecting library file names with the names of functions and variables they define, the features they provide, and the features they require.

Each element is a list and describes one library. The CAR of the list is the absolute file name of the library, as 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.

**(t . fun)** The function *fun* was previously an autoload before this library redefined it as a function. The following element is always **(defun . fun)**, which represents defining *fun* as a function.

**(autoload . fun)**  
The function *fun* was defined as an autoload.

**(require . feature)**  
The feature *feature* was required.

**(provide . feature)**  
The feature *feature* was provided.

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 9.3 [Eval], page 116.

## 15.9 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 `autoload` property of the symbol.)

Before restoring the previous definitions, `unload-feature` runs `remove-hook` to remove functions in the library from certain hooks. These hooks include variables whose names end in ‘hook’ or ‘-hooks’, plus those listed in `unload-feature-special-hooks`. This is to prevent Emacs from ceasing to function because important hooks refer to functions that are no longer defined.

If these measures are not sufficient to prevent malfunction, a library can define an explicit unload hook. If `feature-unload-hook` is defined, it is run as a normal hook before restoring the previous definitions, *instead of* the usual hook-removing actions. The unload hook ought to undo all the global state changes made by the library that might cease to work once the library is unloaded. `unload-feature` can cause problems with libraries that fail to do this, so it should be used with caution.

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.

### 15.10 Hooks for Loading

You can ask for code to be executed if and when a particular library is loaded, by calling `eval-after-load`.

#### `eval-after-load library form`

[Function]

This function arranges to evaluate *form* at the end of loading the file *library*, each time *library* is loaded. If *library* is already loaded, it evaluates *form* right away. Don’t forget to quote *form*!

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:

```
(eval-after-load "edebug" '(def-edebbug-spec c-point t))
```

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`’:

```
(eval-after-load "foo/bar/my_inst.elc" ...)
```

*library* can also be a feature (i.e. a symbol), in which case *form* is evaluated when (`provide library`) is called.

An error in *form* does not undo the load, but does prevent execution of the rest of *form*.

In general, well-designed Lisp programs should not use this feature. The clean and modular ways to interact with a Lisp library are (1) examine and set the library's variables (those which are meant for outside use), and (2) call the library's functions. If you wish to do (1), you can do it immediately—there is no need to wait for when the library is loaded. To do (2), you must load the library (preferably with `require`).

But it is OK to use `eval-after-load` in your personal customizations if you don't feel they must meet the design standards for programs meant for wider use.

**after-load-alist**

[Variable]

This variable, an alist built by `eval-after-load`, holds the expressions to evaluate when particular libraries are loaded. Each element looks like this:

`(regexp-or-feature forms...)`

The key *regexp-or-feature* is either a regular expression or a symbol, and the value is a list of forms. The forms are evaluated when the key matches the absolute true name of the file being loaded or the symbol being provided.

## 16 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.

Compiling a Lisp file with the Emacs byte compiler always reads the file as multibyte text, even if Emacs was started with ‘`--unibyte`’, unless the file specifies otherwise. This is so that compilation gives results compatible with running the same file without compilation. See Section 15.4 [Loading Non-ASCII], page 205.

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; -*-
```

See Section 18.5 [Compilation Errors], page 267, for how to investigate errors occurring in byte compilation.

### 16.1 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 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

(silly-loop 100000)
⇒ ("Fri Mar 18 17:25:57 1994"
    "Fri Mar 18 17:26:28 1994") ; 31 seconds

(byte-compile 'silly-loop)
⇒ [Compiled code not shown]

(silly-loop 100000)
⇒ ("Fri Mar 18 17:26:52 1994"
    "Fri Mar 18 17:26:58 1994") ; 6 seconds
```

In this example, the interpreted code required 31 seconds to run, whereas the byte-compiled code required 6 seconds. These results are representative, but actual results will vary greatly.

## 16.2 The 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`.

The byte compiler produces error messages and warnings about each file in a buffer called ‘\*Compile-Log\*’. These report things in your program that suggest a problem but are not necessarily erroneous.

Be careful when writing macro calls in files that you may someday byte-compile. Macro calls are expanded when they are compiled, so the macros must already be defined for proper compilation. For more details, see Section 13.3 [Compiling Macros], page 177. If a program does not work the same way when compiled as it does when interpreted, erroneous macro definitions are one likely cause (see Section 13.6 [Problems with Macros], page 180). 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.

Normally, compiling a file does not evaluate the file’s contents or load the file. But it does execute any `require` calls at top level in the file. One way to ensure that necessary macro definitions are available during compilation is to require the file that defines them (see Section 15.7 [Named Features], page 209). To avoid loading the macro definition files when someone *runs* the compiled program, write `eval-when-compile` around the `require` calls (see Section 16.5 [Eval During Compile], page 219).

`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; i.e., the compiler does not follow indirection to another symbol. `byte-compile` returns the new, compiled definition of *symbol*.

If *symbol*’s definition is a byte-code function object, `byte-compile` does nothing and returns `nil`. Lisp records only one function definition for any symbol, and if that is already compiled, non-compiled code is not available anywhere. So there is no way to “compile the same definition again.”

```
(defun factorial (integer)
  "Compute factorial of INTEGER."
  (if (= 1 integer) 1
      (* integer (factorial (1- integer)))))
⇒ factorial

(byte-compile 'factorial)
⇒
#[(integer
  "^\301U\203^H^@\301\207\302^H\303^HS!" "\207"
  [integer 1 * factorial]
  4 "Compute factorial of INTEGER."]
```

The result is a byte-code function object. The string it contains is the actual byte-code; each character in it is an instruction or an operand of an instruction. The vector contains all the constants, variable names and function names used by the function, except for certain primitives that are coded as special instructions.

If the argument to `byte-compile` is a `lambda` expression, it 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 compiled.

**byte-compile-file filename &optional load** [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.

If `load` is non-`nil`, this command loads the compiled file after compiling it. Interactively, `load` is the prefix argument.

```
% ls -l push*
-rw-r--r-- 1 lewis      791 Oct  5 20:31 push.el

(byte-compile-file "~/emacs/push.el")
⇒ t

% ls -l push*
-rw-r--r-- 1 lewis      791 Oct  5 20:31 push.el
-rw-rw-rw- 1 lewis      638 Oct  8 20:25 push.elc
```

**byte-recompile-directory directory &optional flag force** [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. 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
```

**byte-code code-string data-vector max-stack** [Function]

This function actually interprets byte-code. A byte-compiled function is actually defined with a body that calls `byte-code`. Don’t call this function yourself—only the byte compiler knows how to generate valid calls to this function.

In Emacs version 18, `byte-code` was always executed by way of a call to the function `byte-code`. Nowadays, `byte-code` is usually executed as part of a byte-code function object, and only rarely through an explicit call to `byte-code`.

### 16.3 Documentation Strings and Compilation

Functions and variables loaded from a byte-compiled file access their documentation strings dynamically from the file whenever needed. This saves space within Emacs, and makes loading faster because the documentation strings themselves need not be processed while loading the file. Actual access to the documentation strings becomes slower as a result, but this normally is not enough to bother users.

Dynamic access to documentation strings does have drawbacks:

- If you delete or move the compiled file after loading it, Emacs can no longer access the documentation strings for the functions and variables in the file.
- If you alter the compiled file (such as by compiling a new version), then further access to documentation strings in this file will probably give nonsense results.

If your site installs Emacs following the usual procedures, these problems will never normally occur. Installing a new version uses a new directory with a different name; as long as the old version remains installed, its files will remain unmodified in the places where they are expected to be.

However, if you have built Emacs yourself and use it from the directory where you built it, you will experience this problem occasionally if you edit and recompile Lisp files. When it happens, you can cure the problem by reloading the file after recompiling it.

You can turn off this feature at compile time by setting `byte-compile-dynamic-docstrings` to `nil`; this is useful mainly if you expect to change the file, and you want Emacs processes that have already loaded it to keep working when the file changes. You can do this globally, or for one source file by specifying a file-local binding for the variable. One way to do that is by adding this string to the file’s first line:

```
--byte-compile-dynamic-docstrings: nil;--
```

**byte-compile-dynamic-docstrings** [Variable]

If this is non-nil, the byte compiler generates compiled files that are set up for dynamic loading of documentation strings.

The dynamic documentation string feature writes compiled files that use a special Lisp reader construct, ‘# $\circ$ count’. This construct skips the next count characters. It also uses the ‘#\$’ construct, which stands for “the name of this file, as a string.” It is usually best not to use these constructs in Lisp source files, since they are not designed to be clear to humans reading the file.

## 16.4 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 becomes much 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.

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*.

## 16.5 Evaluation During Compilation

These features permit you to write code to be evaluated during compilation of a program.

**eval-and-compile** *body...* [Special Form]

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...* [Special Form]

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)) ;;; only macros needed from this
```

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.

## 16.6 Compiler Errors

Byte compilation outputs all errors and warnings into the buffer ‘\*Compile-Log\*’. The messages include file names and line numbers that identify the location of the problem. The usual Emacs commands for operating on compiler diagnostics work properly on these messages.

However, the warnings about functions that were used but not defined are always “located” at the end of the file, so these commands won’t find the places they are really used. To do that, you must search for the function names.

You can suppress the compiler warning for calling an undefined function *func* by conditionalizing the function call 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 a compiler warning for an unbound variable *variable* by conditionalizing its use on a `boundp` test, like this:

```
(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 suppress any compiler warnings using the construct `with-no-warnings`:

`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 this construct around the smallest possible piece of code.

## 16.7 Byte-Code Function Objects

Byte-compiled functions have a special data type: they are *byte-code function objects*.

Internally, a byte-code function object is much like a vector; however, the evaluator handles this data type specially when it appears as a function to be called. The printed representation for a byte-code function object is like that for a vector, with an additional ‘#’ before the opening ‘[’.

A byte-code function object must have at least four elements; there is no maximum number, but only the first six elements have any normal use. They are:

- `arglist` The list of argument symbols.
- `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 24.2 [Accessing Documentation], page 426).

**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`.

```
#[(&optional arg)
  "^\H\204^F@^\S01^P^\S02^H[!^\L207"
  [arg 1 forward-sexp]
  2
  254435
  "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).

You can access the elements of a byte-code object using `aref`; you can also use `vconcat` to create a vector with the same elements.

## 16.8 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 humanly 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 or a byte-code object. 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`. These examples show unoptimized byte-code. Nowadays byte-code is usually optimized, but we did not want to rewrite these examples, since they still serve their purpose.

```
(defun factorial (integer)
  "Compute factorial of an integer."
  (if (= 1 integer) 1
    (* integer (factorial (1- integer)))))

(factorial 4)
⇒ 24

(disassemble 'factorial)
  ;+ byte-code for factorial:
  doc: Compute factorial of an integer.
  args: (integer)

0  constant 1          ; Push 1 onto stack.

1  varref   integer    ; Get value of integer
                       ; from the environment
                       ; and push the value
                       ; onto the stack.

2  eqlsign             ; Pop top two values off stack,
                       ; compare them,
                       ; and push result onto stack.

3  goto-if-nil 10      ; Pop and test top of stack;
                       ; if nil, go to 10,
                       ; else continue.

6  constant 1          ; Push 1 onto top of stack.

7  goto     17          ; Go to 17 (in this case, 1 will be
                       ; returned by the function).

10 constant *
11 varref   integer    ; Push value of integer onto stack.
```

```

12  constant factorial      ; Push factorial onto stack.

13  varref    integer       ; Push value of integer onto stack.

14  sub1          ; Pop integer, decrement value,
                  ; push new value onto stack.

                  ; Stack now contains:
                  ;   - decremented value of integer
                  ;   - factorial
                  ;   - value of integer
                  ;   - *

15  call     1           ; Call function factorial using
                        ;   the first (i.e., the top) element
                        ;   of the stack as the argument;
                        ;   push returned value onto stack.

                  ; Stack now contains:
                  ;   - result of recursive
                  ;       call to factorial
                  ;   - value of integer
                  ;   - *

16  call     2           ; Using the first two
                        ;   (i.e., the top two)
                        ;   elements of the stack
                        ;   as arguments,
                        ;   call the function *,
                        ;   pushing the result onto the stack.

17  return          ; Return the top element
                  ;   of the stack.

⇒ nil

```

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: (n)

0  constant current-time-string ; Push
                                         ;   current-time-string
                                         ;   onto top of stack.

1  call      0                 ; Call current-time-string
                                ;   with no argument,
                                ;   pushing result onto stack.

2  varbind   t1               ; Pop stack and bind t1
                                ;   to popped value.

3  varref    n                ; Get value of n from
                                ;   the environment and push
                                ;   the value onto the stack.

4  sub1                  ; Subtract 1 from top of stack.

5  dup                   ; Duplicate the top of the stack;
                                ;   i.e., copy the top of
                                ;   the stack and push the
                                ;   copy onto the stack.

6  varset    n                ; Pop the top of the stack,
                                ;   and bind n to the value.

                                ; In effect, the sequence dup varset
                                ; 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-nil-else-pop 17 ; Goto 17 if n <= 0
                                ;   (this exits the while loop).
                                ;   else pop top of stack
                                ;   and continue
```

```
12  constant nil          ; Push nil onto stack
   ;   (this is the body of the loop).

13  discard               ; Discard result of the body
   ;   of the loop (a while loop
   ;   is always evaluated for
   ;   its side effects).

14  goto      3           ; Jump back to beginning
   ;   of while loop.

17  discard               ; Discard result of while loop
   ;   by popping top of stack.
   ;   This result is the value nil that
   ;   was not popped by the goto at 9.

18  varref    t1           ; Push value of t1 onto stack.

19  constant current-time-string ; Push
   ;   current-time-string
   ;   onto top of stack.

20  call      0           ; Call current-time-string again.

21  list2                ; Pop top two elements off stack,
   ;   create a list of them,
   ;   and push list onto stack.

22  unbind    1           ; Unbind t1 in local environment.

23  return               ; Return value of the top of stack.

⇒ nil
```

## 17 Advising Emacs Lisp Functions

The *advice* feature lets you add to the existing definition of a function, by *advising the function*. This is a cleaner method for a library to customize functions defined within Emacs—cleaner than redefining the whole function.

Each function can have multiple pieces of advice, separately defined. Each defined piece of advice can be *enabled* or *disabled* explicitly. All the enabled pieces of advice for any given function actually take effect when you activate advice for that function, or when you define or redefine the function. Note that enabling a piece of advice and activating advice for a function are not the same thing.

**Usage Note:** Advice is useful for altering the behavior of existing calls to an existing function. If you want the new behavior for new calls, or for key bindings, you should define a new function (or a new command) which uses the existing function.

**Usage note:** Advising a function can cause confusion in debugging, since people who debug calls to the original function may not notice that it has been modified with advice. Therefore, if you have the possibility to change the code of that function (or ask someone to do so) to run a hook, please solve the problem that way. Advice should be reserved for the cases where you cannot get the function changed.

In particular, this means that a file in Emacs should not put advice on a function in Emacs. There are currently a few exceptions to this convention, but we aim to correct them.

### 17.1 A Simple Advice Example

The command `next-line` moves point down vertically one or more lines; it is the standard binding of `C-n`. When used on the last line of the buffer, this command inserts a newline to create a line to move to if `next-line-add-newlines` is non-`nil` (its default is `nil`.)

Suppose you wanted to add a similar feature to `previous-line`, which would insert a new line at the beginning of the buffer for the command to move to (when `next-line-add-newlines` is non-`nil`). How could you do this?

You could do it by redefining the whole function, but that is not modular. The advice feature provides a cleaner alternative: you can effectively add your code to the existing function definition, without actually changing or even seeing that definition. Here is how to do this:

```
(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))))
```

This expression defines a piece of advice for the function `previous-line`. This piece of advice is named `next-line-at-end`, and the symbol `before` says that it is *before-advice* which should run before the regular definition of `previous-line`. (`&optional arg try-vscroll`) specifies how the advice code can refer to the function's arguments.

When this piece of advice runs, it creates an additional line, in the situation where that is appropriate, but does not move point to that line. This is the correct way to write the advice, because the normal definition will run afterward and will move back to the newly inserted line.

Defining the advice doesn't immediately change the function `previous-line`. That happens when you *activate* the advice, like this:

```
(ad-activate 'previous-line)
```

This is what actually begins to use the advice that has been defined so far for the function `previous-line`. Henceforth, whenever that function is run, whether invoked by the user with *C-p* or *M-x*, or called from Lisp, it runs the advice first, and its regular definition second.

This example illustrates before-advice, which is one *class* of advice: it runs before the function's base definition. There are two other advice classes: *after-advice*, which runs after the base definition, and *around-advice*, which lets you specify an expression to wrap around the invocation of the base definition.

## 17.2 Defining Advice

To define a piece of advice, use the macro `defadvice`. A call to `defadvice` has the following syntax, which is based on the syntax of `defun` and `defmacro`, but adds more:

```
(defadvice function (class name
                         [position] [arglist]
                         [flags...])
  [documentation-string]
  [interactive-form]
  [body-forms...])
```

Here, *function* is the name of the function (or macro or special form) to be advised. From now on, we will write just "function" when describing the entity being advised, but this always includes macros and special forms.

In place of the argument list in an ordinary definition, an advice definition calls for several different pieces of information.

*class* specifies the *class* of the advice—one of `before`, `after`, or `around`. Before-advice runs before the function itself; after-advice runs after the function itself; around-advice is wrapped around the execution of the function itself. After-advice and around-advice can override the return value by setting `ad-return-value`.

**ad-return-value** [Variable]

While advice is executing, after the function's original definition has been executed, this variable holds its return value, which will ultimately be returned to the caller after finishing all the advice. After-advice and around-advice can arrange to return some other value by storing it in this variable.

The argument *name* is the name of the advice, a non-`nil` symbol. The advice name uniquely identifies one piece of advice, within all the pieces of advice in a particular class for a particular *function*. The name allows you to refer to the piece of advice—to redefine it, or to enable or disable it.

The optional *position* specifies where, in the current list of advice of the specified *class*, this new advice should be placed. It should be either `first`, `last` or a number that specifies a zero-based position (`first` is equivalent to 0). If no position is specified, the default is `first`. Position values outside the range of existing positions in this class are mapped to the beginning or the end of the range, whichever is closer. The *position* value is ignored when redefining an existing piece of advice.

The optional *arglist* can be used to define the argument list for the sake of advice. This becomes the argument list of the combined definition that is generated in order to run the advice (see Section 17.10 [Combined Definition], page 235). Therefore, the advice expressions can use the argument variables in this list to access argument values.

The argument list used in advice need not be the same as the argument list used in the original function, but must be compatible with it, so that it can handle the ways the function is actually called. If two pieces of advice for a function both specify an argument list, they must specify the same argument list.

See Section 17.8 [Argument Access in Advice], page 233, for more information about argument lists and advice, and a more flexible way for advice to access the arguments.

The remaining elements, *flags*, are symbols that specify further information about how to use this piece of advice. Here are the valid symbols and their meanings:

- activate** Activate the advice for *function* now. Changes in a function's advice always take effect the next time you activate advice for the function; this flag says to do so, for *function*, immediately after defining this piece of advice.  
This flag has no immediate effect if *function* itself is not defined yet (a situation known as *forward advice*), because it is impossible to activate an undefined function's advice. However, defining *function* will automatically activate its advice.
- protect** Protect this piece of advice against non-local exits and errors in preceding code and advice. Protecting advice places it as a cleanup in an `unwind-protect` form, so that it will execute even if the previous code gets an error or uses `throw`. See Section 10.5.4 [Cleanups], page 133.
- compile** Compile the combined definition that is used to run the advice. This flag is ignored unless `activate` is also specified. See Section 17.10 [Combined Definition], page 235.
- disable** Initially disable this piece of advice, so that it will not be used unless subsequently explicitly enabled. See Section 17.6 [Enabling Advice], page 232.
- preactivate**  
Activate advice for *function* when this `defadvice` is compiled or macroexpanded. This generates a compiled advised definition according to the current advice state, which will be used during activation if appropriate. See Section 17.7 [Preactivation], page 232.  
This is useful only if this `defadvice` is byte-compiled.

The optional *documentation-string* serves to document this piece of advice. When advice is active for *function*, the documentation for *function* (as returned by `documentation`)

combines the documentation strings of all the advice for *function* with the documentation string of its original function definition.

The optional *interactive-form* form can be supplied to change the interactive behavior of the original function. If more than one piece of advice has an *interactive-form*, then the first one (the one with the smallest position) found among all the advice takes precedence.

The possibly empty list of *body-forms* specifies the body of the advice. The body of an advice can access or change the arguments, the return value, the binding environment, and perform any other kind of side effect.

**Warning:** When you advise a macro, keep in mind that macros are expanded when a program is compiled, not when a compiled program is run. All subroutines used by the advice need to be available when the byte compiler expands the macro.

**ad-unadvise** *function* [Command]

This command deletes the advice from *function*.

**ad-unadvise-all** [Command]

This command deletes all pieces of advice from all functions.

### 17.3 Around-Advice

Around-advice lets you “wrap” a Lisp expression “around” the original function definition. You specify where the original function definition should go by means of the special symbol **ad-do-it**. Where this symbol occurs inside the around-advice body, it is replaced with a **progn** containing the forms of the surrounded code. Here is an example:

```
(defadvice foo (around foo-around)
  "Ignore case in 'foo'."
  (let ((case-fold-search t))
    ad-do-it))
```

Its effect is to make sure that case is ignored in searches when the original definition of **foo** is run.

**ad-do-it** [Variable]

This is not really a variable, rather a place-holder that looks like a variable. You use it in around-advice to specify the place to run the function’s original definition and other “earlier” around-advice.

If the around-advice does not use **ad-do-it**, then it does not run the original function definition. This provides a way to override the original definition completely. (It also overrides lower-positioned pieces of around-advice).

If the around-advice uses **ad-do-it** more than once, the original definition is run at each place. In this way, around-advice can execute the original definition (and lower-positioned pieces of around-advice) several times. Another way to do that is by using **ad-do-it** inside of a loop.

## 17.4 Computed Advice

The macro `defadvice` resembles `defun` in that the code for the advice, and all other information about it, are explicitly stated in the source code. You can also create advice whose details are computed, using the function `ad-add-advice`.

`ad-add-advice function advice class position` [Function]

Calling `ad-add-advice` adds `advice` as a piece of advice to `function` in class `class`.

The argument `advice` has this form:

`(name protected enabled definition)`

Here `protected` and `enabled` are flags, and `definition` is the expression that says what the advice should do. If `enabled` is `nil`, this piece of advice is initially disabled (see Section 17.6 [Enabling Advice], page 232).

If `function` already has one or more pieces of advice in the specified `class`, then `position` specifies where in the list to put the new piece of advice. The value of `position` can either be `first`, `last`, or a number (counting from 0 at the beginning of the list). Numbers outside the range are mapped to the beginning or the end of the range, whichever is closer. The `position` value is ignored when redefining an existing piece of advice.

If `function` already has a piece of advice with the same name, then the `position` argument is ignored and the old advice is replaced with the new one.

## 17.5 Activation of Advice

By default, advice does not take effect when you define it—only when you activate advice for the function that was advised. However, the advice will be activated automatically if you define or redefine the function later. You can request the activation of advice for a function when you define the advice, by specifying the `activate` flag in the `defadvice`. But normally you activate the advice for a function by calling the function `ad-activate` or one of the other activation commands listed below.

Separating the activation of advice from the act of defining it permits you to add several pieces of advice to one function efficiently, without redefining the function over and over as each advice is added. More importantly, it permits defining advice for a function before that function is actually defined.

When a function’s advice is first activated, the function’s original definition is saved, and all enabled pieces of advice for that function are combined with the original definition to make a new definition. (Pieces of advice that are currently disabled are not used; see Section 17.6 [Enabling Advice], page 232.) This definition is installed, and optionally byte-compiled as well, depending on conditions described below.

In all of the commands to activate advice, if `compile` is `t` (or anything but `nil` or a negative number), the command also compiles the combined definition which implements the advice. If it is `nil` or a negative number, what happens depends on `ad-default-compilation-action` as described below.

`ad-activate function &optional compile` [Command]

This command activates all the advice defined for `function`.

Activating advice does nothing if *function*'s advice is already active. But if there is new advice, added since the previous time you activated advice for *function*, it activates the new advice.

**ad-deactivate** *function* [Command]

This command deactivates the advice for *function*.

**ad-update** *function* &optional *compile* [Command]

This command activates the advice for *function* if its advice is already activated. This is useful if you change the advice.

**ad-activate-all** &optional *compile* [Command]

This command activates the advice for all functions.

**ad-deactivate-all** [Command]

This command deactivates the advice for all functions.

**ad-update-all** &optional *compile* [Command]

This command activates the advice for all functions whose advice is already activated. This is useful if you change the advice of some functions.

**ad-activate-regexp** *regexp* &optional *compile* [Command]

This command activates all pieces of advice whose names match *regexp*. More precisely, it activates all advice for any function which has at least one piece of advice that matches *regexp*.

**ad-deactivate-regexp** *regexp* [Command]

This command deactivates all pieces of advice whose names match *regexp*. More precisely, it deactivates all advice for any function which has at least one piece of advice that matches *regexp*.

**ad-update-regexp** *regexp* &optional *compile* [Command]

This command activates pieces of advice whose names match *regexp*, but only those for functions whose advice is already activated.

Reactivating a function's advice is useful for putting into effect all the changes that have been made in its advice (including enabling and disabling specific pieces of advice; see Section 17.6 [Enabling Advice], page 232) since the last time it was activated.

**ad-start-advice** [Command]

Turn on automatic advice activation when a function is defined or redefined. This is the default mode.

**ad-stop-advice** [Command]

Turn off automatic advice activation when a function is defined or redefined.

**ad-default-compilation-action** [User Option]

This variable controls whether to compile the combined definition that results from activating advice for a function.

A value of `always` specifies to compile unconditionally. A value of `never` specifies never compile the advice.

A value of `maybe` specifies to compile if the byte-compiler is already loaded. A value of `like-original` specifies to compile the advice if the original definition of the advised function is compiled or a built-in function.

This variable takes effect only if the `compile` argument of `ad-activate` (or any of the above functions) did not force compilation.

If the advised definition was constructed during “preactivation” (see Section 17.7 [Preactivation], page 232), then that definition must already be compiled, because it was constructed during byte-compilation of the file that contained the `defadvice` with the `preactivate` flag.

## 17.6 Enabling and Disabling Advice

Each piece of advice has a flag that says whether it is enabled or not. By enabling or disabling a piece of advice, you can turn it on and off without having to undefine and redefine it. For example, here is how to disable a particular piece of advice named `my-advice` for the function `foo`:

```
(ad-disable-advice 'foo 'before 'my-advice)
```

This function by itself only changes the enable flag for a piece of advice. To make the change take effect in the advised definition, you must activate the advice for `foo` again:

```
(ad-activate 'foo)
```

**ad-disable-advice** *function class name* [Command]

This command disables the piece of advice named *name* in class *class* on *function*.

**ad-enable-advice** *function class name* [Command]

This command enables the piece of advice named *name* in class *class* on *function*.

You can also disable many pieces of advice at once, for various functions, using a regular expression. As always, the changes take real effect only when you next reactivate advice for the functions in question.

**ad-disable-regexp** *regexp* [Command]

This command disables all pieces of advice whose names match *regexp*, in all classes, on all functions.

**ad-enable-regexp** *regexp* [Command]

This command enables all pieces of advice whose names match *regexp*, in all classes, on all functions.

## 17.7 Preactivation

Constructing a combined definition to execute advice is moderately expensive. When a library advises many functions, this can make loading the library slow. In that case, you can use `preactivation` to construct suitable combined definitions in advance.

To use preactivation, specify the `preactivate` flag when you define the advice with `defadvice`. This `defadvice` call creates a combined definition which embodies this piece of advice (whether enabled or not) plus any other currently enabled advice for the same

function, and the function's own definition. If the `defadvice` is compiled, that compiles the combined definition also.

When the function's advice is subsequently activated, if the enabled advice for the function matches what was used to make this combined definition, then the existing combined definition is used, thus avoiding the need to construct one. Thus, pactivation never causes wrong results—but it may fail to do any good, if the enabled advice at the time of activation doesn't match what was used for pactivation.

Here are some symptoms that can indicate that a pactivation did not work properly, because of a mismatch.

- Activation of the advised function takes longer than usual.
- The byte-compiler gets loaded while an advised function gets activated.
- `byte-compile` is included in the value of `features` even though you did not ever explicitly use the byte-compiler.

Compiled pactivated advice works properly even if the function itself is not defined until later; however, the function needs to be defined when you *compile* the pactivated advice.

There is no elegant way to find out why pactivated advice is not being used. What you can do is to trace the function `ad-cache-id-verification-code` (with the function `trace-function-background`) before the advised function's advice is activated. After activation, check the value returned by `ad-cache-id-verification-code` for that function: `verified` means that the pactivated advice was used, while other values give some information about why they were considered inappropriate.

**Warning:** There is one known case that can make pactivation fail, in that a preconstructed combined definition is used even though it fails to match the current state of advice. This can happen when two packages define different pieces of advice with the same name, in the same class, for the same function. But you should avoid that anyway.

## 17.8 Argument Access in Advice

The simplest way to access the arguments of an advised function in the body of a piece of advice is to use the same names that the function definition uses. To do this, you need to know the names of the argument variables of the original function.

While this simple method is sufficient in many cases, it has a disadvantage: it is not robust, because it hard-codes the argument names into the advice. If the definition of the original function changes, the advice might break.

Another method is to specify an argument list in the advice itself. This avoids the need to know the original function definition's argument names, but it has a limitation: all the advice on any particular function must use the same argument list, because the argument list actually used for all the advice comes from the first piece of advice for that function.

A more robust method is to use macros that are translated into the proper access forms at activation time, i.e., when constructing the advised definition. Access macros access actual arguments by position regardless of how these actual arguments get distributed onto the argument variables of a function. This is robust because in Emacs Lisp the meaning of an argument is strictly determined by its position in the argument list.

**ad-get-arg** *position* [Macro]

This returns the actual argument that was supplied at *position*.

**ad-get-args** *position* [Macro]

This returns the list of actual arguments supplied starting at *position*.

**ad-set-arg** *position value* [Macro]

This sets the value of the actual argument at *position* to *value*

**ad-set-args** *position value-list* [Macro]

This sets the list of actual arguments starting at *position* to *value-list*.

Now an example. Suppose the function `foo` is defined as

```
(defun foo (x y &optional z &rest r) ...)
```

and is then called with

```
(foo 0 1 2 3 4 5 6)
```

which means that *x* is 0, *y* is 1, *z* is 2 and *r* is (3 4 5 6) within the body of `foo`. Here is what `ad-get-arg` and `ad-get-args` return in this case:

```
(ad-get-arg 0) ⇒ 0
(ad-get-arg 1) ⇒ 1
(ad-get-arg 2) ⇒ 2
(ad-get-arg 3) ⇒ 3
(ad-get-args 2) ⇒ (2 3 4 5 6)
(ad-get-args 4) ⇒ (4 5 6)
```

Setting arguments also makes sense in this example:

```
(ad-set-arg 5 "five")
```

has the effect of changing the sixth argument to "five". If this happens in advice executed before the body of `foo` is run, then *r* will be (3 4 "five" 6) within that body.

Here is an example of setting a tail of the argument list:

```
(ad-set-args 0 '(5 4 3 2 1 0))
```

If this happens in advice executed before the body of `foo` is run, then within that body, *x* will be 5, *y* will be 4, *z* will be 3, and *r* will be (2 1 0) inside the body of `foo`.

These argument constructs are not really implemented as Lisp macros. Instead they are implemented specially by the advice mechanism.

## 17.9 Advising Primitives

Advising a primitive function (also called a "subr") is risky. Some primitive functions are used by the advice mechanism; advising them could cause an infinite recursion. Also, many primitive functions are called directly from C code. Calls to the primitive from Lisp code will take note of the advice, but calls from C code will ignore the advice.

When the advice facility constructs the combined definition, it needs to know the argument list of the original function. This is not always possible for primitive functions. When advice cannot determine the argument list, it uses (`&rest ad-subr-args`), which always works but is inefficient because it constructs a list of the argument values. You can use `ad-define-subr-args` to declare the proper argument names for a primitive function:

**ad-define-subr-args** *function arglist* [Function]

This function specifies that *arglist* should be used as the argument list for function *function*.

For example,

```
(ad-define-subr-args 'fset '(sym newdef))
```

specifies the argument list for the function *fset*.

## 17.10 The Combined Definition

Suppose that a function has *n* pieces of before-advice (numbered from 0 through *n*-1), *m* pieces of around-advice and *k* pieces of after-advice. Assuming no piece of advice is protected, the combined definition produced to implement the advice for a function looks like this:

```
(lambda arglist
  [ [advised-docstring] [(interactive ...)] ]
  (let (ad-return-value)
    before-0-body-form...
    ...
    before-n-1-body-form...
    around-0-body-form...
    around-1-body-form...
    ...
    around-m-1-body-form...
    (setq ad-return-value
          apply original definition to arglist)
    end-of-around-m-1-body-form...
    ...
    end-of-around-1-body-form...
    end-of-around-0-body-form...
    after-0-body-form...
    ...
    after-k-1-body-form...
    ad-return-value))
```

Macros are redefined as macros, which means adding *macro* to the beginning of the combined definition.

The interactive form is present if the original function or some piece of advice specifies one. When an interactive primitive function is advised, advice uses a special method: it calls the primitive with *call-interactively* so that it will read its own arguments. In this case, the advice cannot access the arguments.

The body forms of the various advice in each class are assembled according to their specified order. The forms of around-advice *l* are included in one of the forms of around-advice *l* - 1.

The innermost part of the around advice onion is

```
apply original definition to arglist
```

whose form depends on the type of the original function. The variable `ad-return-value` is set to whatever this returns. The variable is visible to all pieces of advice, which can access and modify it before it is actually returned from the advised function.

The semantic structure of advised functions that contain protected pieces of advice is the same. The only difference is that `unwind-protect` forms ensure that the protected advice gets executed even if some previous piece of advice had an error or a non-local exit. If any around-advice is protected, then the whole around-advice onion is protected as a result.

# 18 Debugging Lisp Programs

There are three ways to investigate a problem in an Emacs Lisp program, depending on what you are doing with the program when the problem appears.

- If the problem occurs when you run the program, you can use a Lisp debugger to investigate what is happening during execution. In addition to the ordinary debugger, Emacs comes with a source-level debugger, Edebug. This chapter describes both of them.
- If the problem is syntactic, so that Lisp cannot even read the program, you can use the Emacs facilities for editing Lisp to localize it.
- If the problem occurs when trying to compile the program with the byte compiler, you need to know how to examine the compiler’s input buffer.

Another useful debugging tool is the *dribble* file. When a *dribble* file is open, Emacs copies all keyboard input characters to that file. Afterward, you can examine the file to find out what input was used. See Section 39.12 [Terminal Input], page 832.

For debugging problems in terminal descriptions, the `open-terminalscript` function can be useful. See Section 39.13 [Terminal Output], page 834.

## 18.1 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 21.12 [Recursive Editing], page 342.

### 18.1.1 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 frequently cause Lisp errors when invoked inappropriately (such as `C-f` at the end of the buffer), 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 `t`. (The command `toggle-debug-on-error` provides an easy way to do this.)

`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`). If it is `nil`, none call the debugger.

The value can also be a list of error conditions that should call the debugger. For example, if you set it to the list (`void-variable`), then only errors about a variable that has no value invoke the debugger.

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 37 [Processes], page 705.

**debug-ignored-errors**

[User Option]

This variable specifies certain kinds of errors that should not enter the debugger. 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, regardless of the value of `debug-on-error`.

The normal value of this variable lists 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 will need to change this list in order 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, then `debug-on-error` is set to `t` when evaluating with the command `eval-expression`. If `eval-expression-debug-on-error` is `nil`, then the value of `debug-on-error` is not changed. See section “Evaluating Emacs-Lisp Expressions” in *The GNU Emacs Manual*.

**debug-on-signal**

[User Option]

Normally, errors that are caught by `condition-case` never run the debugger, even if `debug-on-error` is non-`nil`. In other words, `condition-case` gets a chance to handle the error before the debugger gets a chance.

If you set `debug-on-signal` to a non-`nil` value, then the debugger gets the first chance at every error; an error will invoke the debugger regardless of any `condition-case`, if it fits the criteria specified by the values of `debug-on-error` and `debug-ignored-errors`.

**Warning:** This variable is strong medicine! Various parts of Emacs handle errors in the normal course of affairs, and you may not even realize that errors happen there. If you set `debug-on-signal` to a non-`nil` value, those errors will enter the debugger.

**Warning:** `debug-on-signal` has no effect when `debug-on-error` is `nil`.

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.

If your init file sets `debug-on-error`, the effect may not last past the end of loading the init file. (This is an undesirable byproduct of the code that implements the ‘`--debug-init`’ command line option.) The best way to make the init file set `debug-on-error` permanently is with `after-init-hook`, like this:

```
(add-hook 'after-init-hook
          (lambda () (setq debug-on-error t)))
```

### 18.1.2 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*.

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`. 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.

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 will probably get enough information to solve the problem.

**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`, then the debugger is not called when you quit. See Section 21.10 [Quitting], page 338.

### 18.1.3 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. It works by inserting the form (`implement-debug-on-entry`) into the function definition as the first form.

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*.

**Warning:** if you redefine a function after using `debug-on-entry` on it, the code to enter the debugger is discarded by the redefinition. In effect, redefining the function cancels the break-on-entry feature for that function.

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* -----


(symbol-function 'fact)
  ⇒ (lambda (n)
    (debug (quote debug))
    (if (zerop n) 1 (* n (fact (1- n)))))

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.

#### 18.1.4 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 10.1 [Sequencing], page 119).

#### 18.1.5 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 21.12 [Recursive Editing], page 342) 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 kills the backtrace buffer.

The backtrace buffer shows you the functions that are executing and their argument values. It also allows you to specify a stack frame by moving point to the line describing that frame. (A stack frame is the place where the Lisp interpreter records information about a particular invocation of a function.) The frame whose line point is on is considered the *current frame*. Some of the debugger commands operate on the current frame. If a line 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.

If a function name is underlined, that means the debugger knows where its source code is located. You can click `Mouse-2` on that name, or move to it and type `RET`, to visit the source code.

The debugger itself must be run byte-compiled, since it makes assumptions about how many stack frames are used for the debugger itself. These assumptions are false if the debugger is running interpreted.

### 18.1.6 Debugger Commands

The debugger buffer (in Debugger mode) provides special commands in addition to the usual Emacs commands. 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.

Here is a list of Debugger mode commands:

- c      Exit the debugger and continue execution. When continuing is possible, it 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).  
Continuing is possible after entry to the debugger due to function entry or exit, explicit invocation, or quitting. You cannot continue if the debugger was entered because of an error.
- 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, 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.
- 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`. **Warning:** if you redefine such a function and thus cancel the effect of `debug-on-entry`, it may erroneously show up in this list.

### 18.1.7 Invoking the Debugger

Here we describe in full detail the function `debug` that is used to invoke the debugger.

**debug &rest debugger-args** [Function]

This function enters the debugger. It switches buffers 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.

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 `debug-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:

<code>lambda</code>	A first argument of <code>lambda</code> means <code>debug</code> was called because of entry to a function when <code>debug-on-next-call</code> was non- <code>nil</code> . The debugger displays ‘Debugger entered--entering a function:’ as a line of text at the top of the buffer.
<code>debug</code>	<code>debug</code> as first argument means <code>debug</code> 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 <code>lambda</code> case. It also marks the stack frame for that function so that it will invoke the debugger when exited.
<code>t</code>	When the first argument is <code>t</code> , this indicates a call to <code>debug</code> due to evaluation of a function call form when <code>debug-on-next-call</code> is non- <code>nil</code> . The debugger displays ‘Debugger entered--beginning evaluation of function call form:’ as the top line in the buffer.
<code>exit</code>	When the first argument is <code>exit</code> , it indicates the exit of a stack frame previously marked to invoke the debugger on exit. The second argument given to <code>debug</code> 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.
<code>error</code>	When the first argument is <code>error</code> , the debugger indicates that it is being entered because an error or <code>quit</code> was signaled and not handled, by displaying ‘Debugger entered--Lisp error:’ followed by the error signaled and any arguments to <code>signal</code> . For example,
	<pre>(let ((debug-on-error t))   (/ 1 0))  ----- Buffer: *Backtrace* ----- Debugger entered--Lisp error: (arith-error)   (/ 1 0) ... ----- Buffer: *Backtrace* -----</pre>
	If an error was signaled, presumably the variable <code>debug-on-error</code> is non- <code>nil</code> . If <code>quit</code> was signaled, then presumably the variable <code>debug-on-quit</code> is non- <code>nil</code> .
<code>nil</code>	Use <code>nil</code> as the first of the <code>debugger-args</code> when you want to enter the debugger explicitly. The rest of the <code>debugger-args</code> 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 <code>debug</code> is called.

### 18.1.8 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 18.1.7 [Invoking the Debugger], page 242).

**backtrace**

[Command]

This function prints a trace of Lisp function calls currently active. This is the function used by `debug` to fill up the ‘\*Backtrace\*’ buffer. It is written in C, since it must have access to the stack to determine which function calls are active. 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 values of the function’s arguments if they are all known; if they are still being computed, the line says so. The arguments of special forms are 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 ...computing arguments...)
(progn ...)
eval((progn (1+ var) (list (quote 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.

**backtrace-frame frame-number**

[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 `frame-number` is out of range, `backtrace-frame` returns `nil`.

## 18.2 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 info on function enter and exit.
- 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 enable you to use it.

### 18.2.1 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 18.2.2 [Instrumenting], page 247, 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 18.2.3 [Edebug Execution Modes], page 247.

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.

### 18.2.2 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`, `eval-current-buffer`, 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`.

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 is not already instrumented. This is possible only if Edebug knows where to find the source for that function; for this reading, 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 18.2.4 [Jumping], page 249), 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; see Section 18.2.15 [Edebug and Macros], page 258, for details.

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.

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-:)`.

If Edebug detects a syntax error while instrumenting, it leaves point at the erroneous code and signals an `invalid-read-syntax` error.

See Section 18.2.9 [Edebug Eval], page 253, for other evaluation functions available inside of Edebug.

### 18.2.3 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.

<i>S</i>	Stop: don't execute any more of the program, but wait for more Edebug commands ( <code>edebug-stop</code> ).
SPC	Step: stop at the next stop point encountered ( <code>edebug-step-mode</code> ).
<i>n</i>	Next: stop at the next stop point encountered after an expression ( <code>edebug-next-mode</code> ). Also see <code>edebug-forward-sexp</code> in Section 18.2.4 [Jumping], page 249.
<i>t</i>	Trace: pause (normally one second) at each Edebug stop point ( <code>edebug-trace-mode</code> ).
<i>T</i>	Rapid trace: update the display at each stop point, but don't actually pause ( <code>edebug-Trace-fast-mode</code> ).
<i>g</i>	Go: run until the next breakpoint ( <code>edebug-go-mode</code> ). See Section 18.2.6.1 [Breakpoints], page 250.
<i>c</i>	Continue: pause one second at each breakpoint, and then continue ( <code>edebug-continue-mode</code> ).
<i>C</i>	Rapid continue: move point to each breakpoint, but don't pause ( <code>edebug-Continue-fast-mode</code> ).
<i>G</i>	Go non-stop: ignore breakpoints ( <code>edebug-Go-nonstop-mode</code> ). You can still stop the program by typing <i>S</i> , 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.

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 (see Section 18.2.16 [Edebug Options], page 263).

When you enter a new Edebug level, the initial execution mode comes from the value of the variable `edebug-initial-mode`. (See Section 18.2.16 [Edebug Options], page 263.) By default, this specifies step 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.

#### `edebug-sit-for-seconds`

[User Option]

This option specifies how many seconds to wait between execution steps in trace mode. The default is 1 second.

### 18.2.4 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 18.2.6.1 [Breakpoints], page 250, 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.
- `i` Step into the function or macro called by the form after point.

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 `C-M-f` 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 `C-M-f` 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`, 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.

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.

### 18.2.5 Miscellaneous Edebug Commands

Some miscellaneous Edebug commands are described here.

- `?` Display the help message for Edebug (`edebug-help`).
- `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 (`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-backtrace`).  
You cannot use debugger commands in the backtrace buffer in Edebug as you would in the standard debugger.  
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-J**. You can display a backtrace of all the pending evaluations with **d**.

### 18.2.6 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.

#### 18.2.6.1 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 18.2.1 [Using Edebug], page 246. 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.
- u** Unset the breakpoint (if any) at the stop point at or after point (`edebug-unset-breakpoint`).

**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 `Go-nonstop`. 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.

### 18.2.6.2 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.

### 18.2.6.3 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`.

### 18.2.7 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 18.2.16 [Edebug Options], page 263.

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 18.2.5 [Edebug Misc], page 249).

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.

### 18.2.8 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.

- 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.

### 18.2.9 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 18.2.14 [The Outside Context], page 257, 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.

**M-: exp RET**  
Evaluate expression *exp* in the context of Edebug itself.

**C-x C-e** Evaluate the expression before point, in the context outside of Edebug (`edebug-eval-last-sexp`).

Edebug supports evaluation of expressions containing references to lexically bound symbols created by the following constructs in ‘`cl.el`’ (version 2.03 or later): `lexical-let`, `macrolet`, and `symbol-macrolet`.

### 18.2.10 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`).

**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* (`(edbug-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 that use variables 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 ‘\*edbug\*’, you can return to the source code buffer with *C-c C-w*. The ‘\*edbug\*’ buffer is killed when you continue execution, and recreated next time it is needed.

### 18.2.11 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 50 if they were `nil`. (Actually, the variables `edbug-print-length` and `edbug-print-level` specify the values to use within Edebug.) See Section 19.6 [Output Variables], page 276.

**edbug-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 '(x y))
      (setcar a a)
```

Custom printing prints this as ‘Result: #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.

Other programs can also use custom printing; see ‘`cust-print.el`’ for details.

### 18.2.12 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 18.2.3 [Edebug Execution Modes], page 247).

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.

### 18.2.13 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 execution 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.

The frequency counts appear as comment lines after each line of code, and 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.

### 18.2.14 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.

#### 18.2.14.1 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` and `max-specpdl-size` are both incremented once to reduce Edebug’s impact on the stack. You could, however, still run out of stack space when using Edebug.
- 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`.

#### 18.2.14.2 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 28.18 [Window Configurations], page 526). When you exit Edebug (by continuing the program), it restores the previous window configuration.

Emacs redisplay 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 18.2.16 [Edebug Options], page 263).

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. If the value of `edebug-save-windows` is a list, only the listed windows are saved and restored.

The window start and horizontal scrolling of the source code buffer are not restored, however, so that the display remains coherent within Edebug.

- 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.

### 18.2.14.3 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 34.6 [Match Data], page 676.
- The variables `last-command`, `this-command`, `last-command-char`, `last-input-char`, `last-input-event`, `last-command-event`, `last-event-frame`, `last-nonmenu-event`, and `track-mouse`. Commands used within 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 `edbug-continue-kbd-macro`.

### 18.2.15 Edebug and Macros

To make Edebug properly instrument expressions that call macros, some extra care is needed. This subsection explains the details.

#### 18.2.15.1 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 13.6.2 [Argument Evaluation], page 180).

```
(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 13.4 [Defining Macros], page 178, for more explanation of the `declare` form.

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.

<code>t</code>	All arguments are instrumented for evaluation.
<code>0</code>	None of the arguments is instrumented.
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. If it is `nil`, the default, none of the arguments is instrumented for evaluation. If it is non-`nil`, all arguments are instrumented.

### 18.2.15.2 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. See Section 18.2.15.3 [Backtracking], page 262, for more details.

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 18.2.15.4 [Specification Examples], page 263, for the referenced examples):

<b>sexp</b>	A single unevaluated Lisp object, which is not instrumented.
<b>form</b>	A single evaluated expression, which is instrumented.
<b>place</b>	A place to store a value, as in the Common Lisp <code>setf</code> construct.
<b>body</b>	Short for <code>&amp;rest form</code> . See <code>&amp;rest</code> below.
<b>function-form</b>	A function form: either a quoted function symbol, a quoted lambda expression, or a form (that should evaluate to a function symbol or lambda expression). This is useful when an argument that's a lambda expression might be quoted with <code>quote</code> rather than <code>function</code> , since it instruments the body of the lambda expression either way.
<b>lambda-expr</b>	A lambda expression with no quoting.
<b>&amp;optional</b>	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 <code>[&amp;optional specs...]</code> . To specify that several elements must all match or none, use <code>&amp;optional [specs...]</code> . See the <code>defun</code> example.
<b>&amp;rest</b>	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 <code>[&amp;rest specs...]</code> . To specify several elements that must all match on every repetition, use <code>&amp;rest [specs...]</code> .
<b>&amp;or</b>	Each of the following elements in the specification list is an alternative. One of the alternatives must match, or the <code>&amp;or</code> specification fails.  Each list element following <code>&amp;or</code> is a single alternative. To group two or more list elements as a single alternative, enclose them in <code>[...]</code> .
<b>&amp;not</b>	Each of the following elements is matched as alternatives as if by using <code>&amp;or</code> , but if any of them match, the specification fails. If none of them match, nothing is matched, but the <code>&amp;not</code> specification succeeds.
<b>&amp;define</b>	Indicates that the specification is for a 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 <code>&amp;define</code> keyword should be the first element in a list specification.
<b>nil</b>	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.
<b>gate</b>	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 18.2.15.3 [Backtracking], page 262, for more details. Also see the `let` example.

**`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-edbug-spec` just as for macros. See the `defun` example.

Otherwise, the symbol should be a predicate. The predicate is called with the argument and the specification fails if the predicate returns `nil`. In either case, that 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`**

The argument, a symbol, is the name of the defining form.

A defining form is not required to have a name field; and it may have multiple name fields.

<b>:name</b>	This construct does not actually match an argument. The element following <b>:name</b> should be a symbol; it is used as an additional name component for the definition. You can use this to add a unique, static component to the name of the definition. It may be used more than once.
<b>arg</b>	The argument, a symbol, is the name of an argument of the defining form. However, lambda-list keywords (symbols starting with ‘&’) are not allowed.
<b>lambda-list</b>	This matches a lambda list—the argument list of a lambda expression.
<b>def-body</b>	The argument is the body of code in a definition. This is like <b>body</b> , described above, but a definition body must be instrumented with a different Edebug call that looks up information associated with the definition. Use <b>def-body</b> for the highest level list of forms within the definition.
<b>def-form</b>	The argument is a single, highest-level form in a definition. This is like <b>def-body</b> , except use this to match a single form rather than a list of forms. As a special case, <b>def-form</b> also means that tracing information is not output when the form is executed. See the <b>interactive</b> example.

### 18.2.15.3 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; it is reenabled 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 or string specification, 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.

#### 18.2.15.4 Specification Examples

It may be easier to understand Edebug specifications by studying the examples provided here.

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 `defmacro` 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.

```
(def-edebug-spec defmacro defun) ; Indirect ref to defun spec.
(def-edebug-spec defun
  (&define name lambda-list
    [&optional stringp] ; Match the doc string, if present.
    [&optional ("interactive" interactive)]
    def-body))

(def-edebug-spec lambda-list
  (([&rest arg]
    [&optional [&optional "arg" &rest arg]]
    &optional [&rest "arg"]
    )))

(def-edebug-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 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-spec backquote-form
  (&or ([&or "," ",@"] &or ("quote" backquote-form) form)
       (backquote-form . [&or nil backquote-form])
       (vector &rest backquote-form)
       sexp))
```

#### 18.2.16 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 `edebug-setup-hook` is reset 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 18.2.2 [Instrumenting], page 247.

**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`, `eval-buffer`, and `eval-current-buffer`.

Use the command `M-x edebug-all-defs` to toggle the value of this option. See Section 18.2.2 [Instrumenting], page 247.

**edebug-all-forms** [User Option]

If this is non-`nil`, the commands `eval-defun`, `eval-region`, `eval-buffer`, and `eval-current-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 18.2.2 [Instrumenting], page 247.

**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`.

If the value 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 18.2.14.2 [Edebug Display Update], page 257.

**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 which 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 18.2.14.2 [Edebug Display Update], page 257.

**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`. See Section 18.2.3 [Edebug Execution Modes], page 247.

**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 18.2.12 [Trace Buffer], page 255.

**edebug-test-coverage** [User Option]

If non-`nil`, Edebug tests coverage of all expressions debugged. See Section 18.2.13 [Coverage Testing], page 256.

**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 18.2.3 [Edebug Execution Modes], page 247.

**edebug-on-error** [User Option]

Edebug binds `debug-on-error` to this value, if `debug-on-error` was previously `nil`. See Section 18.2.7 [Trapping Errors], page 252.

**edebug-on-quit** [User Option]

Edebug binds `debug-on-quit` to this value, if `debug-on-quit` was previously `nil`. See Section 18.2.7 [Trapping Errors], page 252.

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 18.2.6.2 [Global Break Condition], page 251.

## 18.3 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` 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.)

### 18.3.1 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`. 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` 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` 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-\_*, 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.

### 18.3.2 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* to find the end of the first unbalanced defun.

Then find the actual matching close parenthesis by typing *C-M-f* 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* 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-\_*, 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.

## 18.4 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 18.2.13 [Coverage Testing], page 256). These features partly duplicate each other, and it would be cleaner to combine them.

## 18.5 Debugging Problems in Compilation

When an error happens during byte compilation, it is normally due to invalid syntax in the program you are compiling. The compiler prints a suitable error message in the ‘\*Compile-Log\*’ buffer, and then stops. The message may state a function name in which the error was found, or it may not. Either way, here is how to find out where in the file the error occurred.

What you should do is switch to the buffer ‘\*Compiler Input\*’. (Note that the buffer name starts with a space, so it does not show up in *M-x list-buffers*.) This buffer contains the program being compiled, and point shows how far the byte compiler was able to read.

If the error was due to invalid Lisp syntax, point shows exactly where the invalid syntax was *detected*. The cause of the error is not necessarily near by! Use the techniques in the previous section to find the error.

If the error was detected while compiling a form that had been read successfully, then point is located at the end of the form. In this case, this technique can’t localize the error precisely, but can still show you which function to check.

# 19 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).

## 19.1 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.

## 19.2 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 <i>buffer</i> , starting with the character directly after point. Point advances as characters are read.   |
| marker | The input characters are read from the buffer that <i>marker</i> 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. |

<i>string</i>	The input characters are taken from <i>string</i> , starting at the first character in the string and using as many characters as required.
<i>function</i>	The input characters are generated by <i>function</i> , which must support two kinds of calls: <ul style="list-style-type: none"> <li>• When it is called with no arguments, it should return the next character.</li> <li>• When it is called with one argument (always a character), <i>function</i> should save the argument and arrange to return it on the next call. This is called <i>unreading</i> 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 <i>function</i> returns.</li> </ul>
<i>t</i>	<i>t</i> 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, standard input is used instead of the minibuffer. For example,
	(message "%s" (read t))
	will read a Lisp expression from standard input and print the result to standard output.
<i>nil</i>	<i>nil</i> supplied as an input stream means to use the value of <b>standard-input</b> instead; that value is the <i>default input stream</i> , and must be a non- <i>nil</i> input stream.
<i>symbol</i>	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`.

### `get-file-char` [Function]

This function is used internally as an input stream to read from the input file opened by the function `load`. Don’t use this function yourself.

### 19.3 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)
```

**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.

### 19.4 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.

<i>marker</i>	The output characters are inserted into the buffer that <i>marker</i> 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).
<i>function</i>	The output characters are passed to <i>function</i> , 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.
<i>t</i>	The output characters are displayed in the echo area.
<i>nil</i>	<i>nil</i> specified as an output stream means to use the value of <b>standard-output</b> instead; that value is the <i>default output stream</i> , and must not be <i>nil</i> .
<i>symbol</i>	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 67). 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.

## 19.5 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.) 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`

[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"
```

**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** [Function]

This function outputs a newline to *stream*. The name stands for “terminate print.”

**write-char character &optional stream** [Function]

This function outputs *character* to *stream*. It returns *character*.

**prin1-to-string object &optional noescape** [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 *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 56, 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`”.

## 19.6 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. `(quote foo)` prints as `'foo`, `(function foo)` as `#'foo`, and backquoted forms print using modern backquote syntax.

### **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-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-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.

**print-gensym**

[Variable]

If non-`nil`, this variable enables detection of uninterned symbols (see Section 8.3 [Creating Symbols], page 104) 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. Its default value 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.

## 20 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 38.4 [The Echo Area], page 741), but only while it is in use for reading an argument.

### 20.1 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 29.8 [Minibuffers and Frames], page 543.)

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 32.19.11 [Fields], page 630), 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. (In older Emacs versions, the prompt was displayed using a special mechanism and was not part of the buffer contents.)

The minibuffer’s window is normally a single line; it grows automatically if necessary if the contents require more space. You can explicitly resize it temporarily with the window sizing commands; it reverts to its normal size when the minibuffer is exited. You can resize it permanently by using the window sizing commands in the frame’s other window, when the minibuffer is not active. If the frame contains just a minibuffer, you can change the minibuffer’s 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 21.4 [Command Loop Info], page 312). Your program should bind them around the code that uses the minibuffer, if you do not want that to change them.

If a command uses a minibuffer while there is an active minibuffer, this is called a *recursive minibuffer*. The first minibuffer is named ‘*\*Minibuf-0\**’. 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. 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 20.13 [Recursive Mini], page 302).

Like other buffers, a minibuffer uses a local keymap (see Chapter 22 [Keymaps], page 347) 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 20.2 [Text from Minibuffer], page 279,

for the non-completion minibuffer local maps. See Section 20.6.3 [Completion Commands], page 289, for the minibuffer local maps for completion.

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.

## 20.2 Reading Text Strings with the Minibuffer

Most often, the minibuffer is used to read text as a string. It can also be used to read a Lisp object in textual form. The most basic primitive for minibuffer input is `read-from-minibuffer`; it can do either one. There are also specialized commands for reading commands, variables, file names, etc. (see Section 20.6 [Completion], page 285).

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 21.2 [Defining Commands], page 305.

**`read-from-minibuffer`** *prompt-string* &**`optional`** *initial-contents* [Function]  
*keymap* *read* *hist* *default* *inherit-input-method*

This function is the most general way to get input through 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 19.3 [Input Functions], page 271).

The first thing this function does is to activate a minibuffer and display it with *prompt-string* as the prompt. This value must be a string. 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 19.3 [Input Functions], page 271, for information about reading.)

The argument *default* specifies a default value to make available through the history commands. It should be a string, or `nil`. If non-`nil`, the user can access it using `next-history-element`, usually bound in the minibuffer to `M-n`. If *read* is non-`nil`, then *default* is also used as the input to *read*, if the user enters empty input. (If *read* is non-`nil` and *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 is different 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 *hist* specifies which history list variable to use for saving the input and for history commands used in the minibuffer. It defaults to `minibuffer-history`. See Section 20.4 [Minibuffer History], page 282.

If the variable `minibuffer-allow-text-properties` is non-`nil`, then the string which is returned includes whatever text properties were present in the minibuffer. Otherwise all the text properties are stripped when the value is returned.

If the argument `inherit-input-method` is non-`nil`, then the minibuffer inherits the current input method (see Section 33.11 [Input Methods], page 659) and the setting of `enable-multibyte-characters` (see Section 33.1 [Text Representations], page 640) from whichever buffer was current before entering the minibuffer.

Use of `initial-contents` is mostly deprecated; we recommend using a non-`nil` value only in conjunction with specifying a cons cell for `hist`. See Section 20.5 [Initial Input], page 284.

**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, or `nil`, which is equivalent to an empty string.

This function is a simplified interface to 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)
      default
      value))
```

**minibuffer-allow-text-properties** [Variable]

If this variable is `nil`, then **read-from-minibuffer** strips all text properties from the minibuffer input before returning it. This variable also affects **read-string**. However, **read-no-blanks-input** (see below), as well as **read-minibuffer** and related functions (see Section 20.3 [Reading Lisp Objects With the Minibuffer], page 281), and all functions that do minibuffer input with completion, discard text properties unconditionally, regardless of the value of this variable.

**minibuffer-local-map** [Variable]

This is the default local keymap for reading from the minibuffer. By default, it makes the following bindings:

<i>C-j</i>	<code>exit-minibuffer</code>
<code>RET</code>	<code>exit-minibuffer</code>
<i>C-g</i>	<code>abort-recursive-edit</code>
<i>M-n</i>	
<code>DOWN</code>	<code>next-history-element</code>

<i>M-p</i>	
UP	previous-history-element
<i>M-s</i>	next-matching-history-element
<i>M-r</i>	previous-matching-history-element

**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

## 20.3 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, and then evaluates it. 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 19.5 [Output Functions], page 273.

The first thing **edit-and-eval-command** does is to activate the minibuffer with *prompt* as the prompt. Then it inserts the printed representation of *form* in the minibuffer, and lets the user edit it. When the user exits the minibuffer, the edited text is read with **read** and then evaluated. The resulting value becomes the value of **edit-and-eval-command**.

In the following example, we offer the user an expression with initial text which is a valid form already:

```
(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. **edit-and-eval-command** returns **nil** in this example.

## 20.4 Minibuffer History

A *minibuffer history list* records previous minibuffer inputs so the user can reuse them conveniently. A history list is actually a symbol, not a list; it is a variable whose value is a list of strings (previous inputs), most recent first.

There are many separate 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 the history list with the optional *hist* argument to either **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 20.5 [Initial Input], page 284).

If you don't specify *hist*, 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.

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. 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.

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**.

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**. By default, **history-add-new-input** is set to a non-**nil** value.

**history-length**

[Variable]

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 no maximum (don't delete old elements). The value of **history-length** property of the history list variable's symbol, if set, overrides this variable for that particular history list.

**history-delete-duplicates** [Variable]  
 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.

## 20.5 Initial Input

Several of the functions for minibuffer input have an argument called *initial* or *initial-contents*. 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 *hist* or *history* argument. See Section 20.4 [Minibuffer History], page 282.

*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 in user code.

## 20.6 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.

### 20.6.1 Basic Completion Functions

The completion functions `try-completion`, `all-completions` and `test-completion` have nothing in themselves to do with minibuffers. We describe them in this chapter so as to keep them near the higher-level completion features that do use the minibuffer.

If you store a completion alist in a variable, you should mark the variable as “risky” with a non-nil `risky-local-variable` property.

**try-completion** *string collection &optional predicate* [Function]

This function returns the longest common substring of all possible completions of *string* in *collection*. The value of *collection* must be a list of strings or symbols, an alist, an obarray, a hash table, or a function that implements a virtual set of strings (see below).

Completion compares *string* against each of the permissible completions specified by *collection*; if the beginning of the permissible completion equals *string*, it matches. If no permissible completions match, `try-completion` returns `nil`. If only one permissible completion matches, and the match is exact, then `try-completion` returns `t`. Otherwise, the value is the longest initial sequence common to all the permissible completions that match.

If *collection* is an alist (see Section 5.8 [Association Lists], page 81), the permissible completions are the elements of the alist that are either strings, symbols, or conses whose CAR is a string or symbol. Symbols are converted to strings using `symbol-name`. Other elements of the alist are ignored. (Remember that in Emacs Lisp, the

elements of alists do not *have* to be conses.) In particular, a list of strings or symbols is allowed, even though we usually do not think of such lists as alists.

If *collection* is an obarray (see Section 8.3 [Creating Symbols], page 104), the names of all symbols in the obarray form the set of permissible completions. The global variable `obarray` holds an obarray containing the names of all interned Lisp symbols.

Note that the only valid way to make a new obarray is to create it empty and then add symbols to it one by one using `intern`. Also, you cannot intern a given symbol in more than one obarray.

If *collection* is a hash table, then the keys that are strings are the possible completions. Other keys are ignored.

You can also use a symbol that is 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 reason for 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 20.6.6 [Programmed Completion], page 296.

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 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 nospace* [Function]

This function returns a list of all possible completions of *string*. The arguments to this function (aside from *nospace*) are the same as those of **try-completion**. Also, this function uses **completion-regexp-list** in the same way that **try-completion** does. The optional argument *nospace* only matters if *string* is the empty string. In that case, if *nospace* is non-*nil*, completions that start with a space are ignored.

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 20.6.6 [Programmed Completion], page 296.

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 possibility 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-ignore-case** [Variable]

If the value of this variable is non-*nil*, Emacs does not consider case significant in completion.

**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 34.2 [Searching and Case], page 663) bound to the value of `completion-ignore-case`.

**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 of use:

```
(defvar foo (lazy-completion-table foo make-my-alist))
```

## 20.6.2 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* [Function]  
*initial* *hist* *default* *inherit-input-method*

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 *collection* and *predicate* to the function `try-completion`. 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 287.

If *require-match* is `nil`, the exit commands work regardless of the input in the minibuffer. If *require-match* is `t`, the usual minibuffer exit commands won't exit unless the input completes to an element of *collection*. If *require-match* is neither `nil` nor `t`, then the exit commands won't exit unless the input already in the buffer matches an element of *collection*.

However, empty input is always permitted, regardless of the value of *require-match*; in that case, `completing-read` returns *default*, or "", if *default* is `nil`. The value of *default* (if non-`nil`) is also available to the user through the history commands.

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 20.6.3 [Completion Commands], page 289.

The argument *hist* specifies which history list variable to use for saving the input and for minibuffer history commands. It defaults to `minibuffer-history`. See Section 20.4 [Minibuffer History], page 282.

The argument *initial* is mostly deprecated; we recommend using a non-`nil` value only in conjunction with specifying a cons cell for *hist*. See Section 20.5 [Initial Input], page 284. 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 33.11 [Input Methods], page 659) and the setting of `enable-multibyte-characters` (see Section 33.1 [Text Representations], page 640) from whichever buffer was current before entering the minibuffer.

If the built-in variable `completion-ignore-case` is non-`nil`, completion ignores case when comparing the input against the possible matches. See Section 20.6.1 [Basic Completion], page 285. 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.

### 20.6.3 Minibuffer Commands that Do Completion

This section describes the keymaps, commands and user options used in the minibuffer to do completion. The description refers to the situation when Partial Completion mode is disabled (as it is by default). When enabled, this minor mode uses its own alternatives to some of the commands described below. See section “Completion Options” in *The GNU Emacs Manual*, for a short description of Partial Completion mode.

#### `minibuffer-completion-table`

[Variable]

The value of this variable is the collection used for completion in the minibuffer. This is the global variable that contains what `completing-read` passes to `try-completion`. It is used by minibuffer completion commands such as `minibuffer-complete-word`.

#### `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]

When the value of this variable is non-`nil`, Emacs asks for confirmation of a completion before exiting the minibuffer. `completing-read` binds this variable, and the function `minibuffer-complete-and-exit` checks the value before exiting.

#### `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 35 [Syntax Tables], page 684.

**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 &optional common-substring* [Function]

This function displays *completions* to the stream in `standard-output`, usually a buffer. (See Chapter 19 [Read and Print], page 268, 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.

The argument *common-substring* is the prefix that is common to all the completions. With normal Emacs completion, it is usually the same as the string that was completed. `display-completion-list` uses this to highlight text in the completion list for better visual feedback. This is not needed in the minibuffer; for minibuffer completion, you can pass `nil`.

This function is called by `minibuffer-completion-help`. The most 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)
    (buffer-string)))
```

**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:

?	<code>minibuffer-completion-help</code>
SPC	<code>minibuffer-complete-word</code>

TAB minibuffer-complete

with other characters bound as in `minibuffer-local-map` (see [Definition of `minibuffer-local-map`], page 280).

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:

with other characters bound as in `minibuffer-local-map`.

`minibuffer-local-filename-completion-map`

[Variable]

This is like `minibuffer-local-completion-map` except that it does not bind SPC. This keymap is used by the function `read-file-name`.

minibuffer-local-must-match-filename-map

## [Variable]

This is like `minibuffer-local-must-match-map` except that it does not bind SPC. This keymap is used by the function `read-file-name`.

#### 20.6.4 High-Level Completion Functions

This section describes the higher-level convenient 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 21.2 [Defining Commands], page 305.

**read-buffer** *prompt* &**optional** *default* *existing*

## [Function]

This function reads the name of a buffer and returns it as a string. 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 or a buffer. 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 860).

If `existing` is non-`nil`, then the name specified must be that of an existing buffer. The usual commands to exit the minibuffer do not exit if the text is not valid, and RET does completion to attempt to find a valid name. If `existing` is neither `nil` nor `t`, confirmation is required after completion. (However, `default` is not checked for validity; it is returned, whatever it is, if the user exits with the minibuffer empty.)

In the following example, the user enters ‘`minibuffer.t`’, and then types RET. The argument `existing` 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**

[Variable]

This variable specifies how to read buffer names. For example, if you set this variable to `iswitchb-read-buffer`, all Emacs commands that call `read-buffer` to read a buffer name will actually use the `iswitchb` package to read it.

**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 21.3 [Interactive Call], page 310.

The argument `default` specifies what to return if the user enters null input. It can be a symbol or a string; if it is a string, `read-command` interns it before returning it. 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.

```
(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 user variable and returns it as a symbol.

The argument *default* specifies what to return if the user enters null input. It can be a symbol or a string; if it is a string, **read-variable** interns it before returning it. If *default* is `nil`, that means no default has been specified; then if the user enters null input, the return value is (**intern** "").

```
(read-variable "Variable name? ")

;; After evaluation of the preceding expression,
;; the following prompt appears,
;; with an empty minibuffer:

----- Buffer: Minibuffer -----
Variable name? *
----- Buffer: Minibuffer -----
```

If the user then types *fill-p RET*, **read-variable** returns **fill-prefix**.

In general, **read-variable** is similar to **read-command**, but uses the predicate **user-variable-p** instead of **commandp**:

```
(read-variable prompt)
≡
(intern
  (completing-read prompt obarray
    'user-variable-p t nil))
```

See also the functions **read-coding-system** and **read-non-nil-coding-system**, in Section 33.10.4 [User-Chosen Coding Systems], page 652, and **read-input-method-name**, in Section 33.11 [Input Methods], page 659.

## 20.6.5 Reading File Names

Here is another high-level completion function, designed for reading a file name. It provides special features including automatic insertion of the default directory.

**read-file-name** *prompt* &**optional** *directory* *default* *existing* *initial* [Function]  
*predicate*

This function reads a file name in the minibuffer, prompting with *prompt* and providing completion.

If *existing* is `non-nil`, then the user must specify the name of an existing file; RET performs completion to make the name valid if possible, and then refuses to exit if it is not valid. If the value of *existing* is neither `nil` nor `t`, then RET also requires confirmation after completion. If *existing* is `nil`, then the name of a nonexistent file is acceptable.

**read-file-name** uses **minibuffer-local-filename-completion-map** as the keymap if *existing* is `nil`, and uses **minibuffer-local-must-match-filename-map** if *existing* is `non-nil`. See Section 20.6.3 [Completion Commands], page 289.

The argument *directory* specifies the directory to use for completion of relative file names. It should be an absolute directory name. If **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`. **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 *existing*. However, if *existing* 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 *existing*. 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 possibilities. A file name is an acceptable value if *predicate* returns non-*nil* for it.

`read-file-name` does not automatically expand file names. You must call `expand-file-name` yourself if an absolute file name is required.

Here is an example:

```
(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** [Variable]

If this variable is non-nil, `read-file-name` ignores case when performing completion.

**read-directory-name** *prompt &optional directory default existing initial* [Function]

This function is like `read-file-name` but allows only directory names as completion possibilities.

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 21.2.2 [Code Characters for interactive], page 307.) 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 -----
```

### 20.6.6 Programmed Completion

Sometimes it is not possible to create an alist or an obarray containing all the intended possible completions. In such a case, you can supply your own function to compute the completion of a given string. This is called *programmed completion*.

To use this feature, pass a symbol with a function definition as the *collection* argument to `completing-read`. The function `completing-read` arranges to pass your completion function along to `try-completion` and `all-completions`, which will then let your function do all the work.

The completion function should accept three arguments:

- The string to be completed.
- The predicate function to filter possible matches, or `nil` if none. Your function should call the predicate for each possible match, and ignore the possible match if the predicate returns `nil`.
- A flag specifying the type of operation.

There are three flag values for three operations:

- `nil` specifies `try-completion`. The completion function should return the completion of the specified string, or `t` if the string is a unique and exact match already, or `nil` if the string matches no possibility.

If the string is an exact match for one possibility, but also matches other longer possibilities, the function should return the string, not `t`.

- `t` specifies `all-completions`. The completion function should return a list of all possible completions of the specified string.
- `lambda` specifies `test-completion`. The completion function should return `t` if the specified string is an exact match for some possibility; `nil` otherwise.

It would be consistent and clean for completion functions to allow lambda expressions (lists that are functions) as well as function symbols as *collection*, but this is impossible. Lists as completion tables already have other meanings, and it would be unreliable to treat one differently just because it is also a possible function. So you must arrange for any function you wish to use for completion to be encapsulated in a symbol.

Emacs uses programmed completion when completing file names. See Section 25.8.6 [File Name Completion], page 461.

#### `dynamic-completion-table` function

[Macro]

This macro is a convenient way to write a function that can act as programmed completion function. The argument *function* should be a function that takes one argument, a string, and returns an alist of possible completions of it. You can think of `dynamic-completion-table` as a transducer between that interface and the interface for programmed completion functions.

## 20.7 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—more precisely, if `last-nonmenu-event` (see Section 21.4 [Command Loop Info], page 312) is either `nil` or a list—then it uses a dialog box or pop-up menu to ask the question. Otherwise, it uses keyboard input. You can force use of the mouse or use of keyboard input by binding `last-nonmenu-event` to a suitable value around the call.

Strictly speaking, `yes-or-no-p` uses the minibuffer and `y-or-n-p` does not; but it seems best to describe them together.

**`y-or-n-p` *prompt*** [Function]

This function asks the user a question, expecting input in the echo area. 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-J` to mean “quit,” like `C-g`, because the question might look like a minibuffer and for that reason the user might try to use `C-J` 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 echo area, 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 does not actually use the minibuffer, since it does not allow editing of the answer. It actually uses the echo area (see Section 38.4 [The Echo Area], page 741), which uses the same screen space as the minibuffer. The cursor moves to the echo area while the question is being asked.

The answers and their meanings, even ‘`y`’ and ‘`n`’, are not hardwired. The keymap `query-replace-map` specifies them. See Section 34.7 [Search and Replace], page 681.

In the following example, the user first types `q`, which is invalid. At the next prompt the user types `y`.

```
(y-or-n-p "Do you need a lift? ")

;; After evaluation of the preceding expression,
;; the following prompt appears in the echo area:

----- Echo area -----
Do you need a lift? (y or n)
----- Echo area -----


;; If the user then types q, the following appears:

----- Echo area -----
Please answer y or n. Do you need a lift? (y or n)
----- Echo area -----


;; When the user types a valid answer,
;; it is displayed after the question:

----- Echo area -----
Do you need a lift? (y or n) y
----- Echo area -----
```

We show successive lines of echo area messages, but only one actually appears on the screen at a time.

**y-or-n-p-with-timeout** *prompt seconds default-value* [Function]

Like **y-or-n-p**, except that if the user fails to answer within *seconds* seconds, this function stops waiting and returns *default-value*. It works by setting up a timer; see Section 39.10 [Timers], page 829. The argument *seconds* may be an integer or a floating point 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 echo area, 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 -----
```

## 20.8 Asking Multiple Y-or-N 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 should expect no arguments, and should return either the next object to ask about, or **nil** meaning 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*).

If not a string, *prompter* should be a function of one argument (the next object to ask about) and should return the question text. If the value is a string, that is the question to ask the user. The function can also return `t` meaning do act on this object (and don't ask the user), or `nil` meaning ignore this object (and don't ask the user).

The argument *actor* says how to act on the answers that the user gives. It should be a function of one argument, and it is called with each object that the user says yes for. Its argument is always an object obtained from *list*.

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 the objects conceptually being acted on, *plural* is the corresponding plural noun, and *action* is a transitive verb describing what *actor* does.

If you don't specify *help*, the default is ("object" "objects" "act on").

Each time a question is asked, the user may enter `y`, `Y`, or `SPC` to act on that object; `n`, `N`, or `DEL` to skip that object; `!` to act on all following objects; `ESC` or `q` to exit (skip all following objects); `.` (period) to act on the current object and then exit; or `C-h` to 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 34.7 [Search and Replace], page 681.

You can use `action-alist` to specify additional possible answers and what they mean. It is an alist of elements of the form (`char function help`), each of which defines one additional answer. In this element, `char` is a character (the answer); `function` is a function of one argument (an object from *list*); `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 “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.

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—more precisely, if `last-nonmenu-event` (see Section 21.4 [Command Loop Info], page 312) is either `nil` or a list—then it uses a dialog box or pop-up menu to ask the question. In this case, it does not use keyboard input or the echo area. You can force use of the mouse or use of keyboard input by binding `last-nonmenu-event` to a suitable value around the call.

The return value of `map-y-or-n-p` is the number of objects acted on.

## 20.9 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.

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.

## 20.10 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.

**self-insert-and-exit** [Command]

This command exits the active minibuffer after inserting the last character typed on the keyboard (found in `last-command-char`; see Section 21.4 [Command Loop Info], page 312).

**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.

**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).

## 20.11 Minibuffer Windows

These functions access and select minibuffer windows and test whether they are active.

**active-minibuffer-window** [Function]

This function returns the currently active minibuffer window, or `nil` if none is currently active.

**minibuffer-window &optional frame** [Function]

This function returns the minibuffer window used for frame *frame*. If *frame* is `nil`, that stands for the current 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.

**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 current frame.

**window-minibuffer-p** &optional *window* [Function]

This function returns non-*nil* if *window* is a minibuffer window. *window* defaults to the selected window.

It is not correct to determine whether a given window is a 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*, assumed to be a minibuffer window, is currently active.

## 20.12 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 32.19 [Text Properties], page 615.

**minibuffer-completion-contents** [Function]

This is like **minibuffer-contents**, except that it returns only the contents before point. That is the part that completion commands operate on. See Section 20.6.2 [Minibuffer Completion], page 288.

**delete-minibuffer-contents** [Function]

This function 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.

## 20.13 Recursive Minibuffers

These functions and variables deal with recursive minibuffers (see Section 21.12 [Recursive Editing], page 342):

### `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 window is active. Such invocation produces a recursive editing level for a new minibuffer. The outer-level minibuffer is invisible while you are editing the inner one.

If this variable is `nil`, you cannot invoke minibuffer commands when the minibuffer window 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 21.2.1 [Using Interactive], page 305). The minibuffer command `next-matching-history-element` (normally `M-s` in the minibuffer) does the latter.

## 20.14 Minibuffer Miscellany

### `minibufferp &optional buffer-or-name`

[Function]

This function returns `non-nil` if `buffer-or-name` is a minibuffer. If `buffer-or-name` is omitted, it tests the current buffer.

### `minibuffer-setup-hook`

[Variable]

This is a normal hook that is run whenever the minibuffer is entered. See Section 23.1 [Hooks], page 382.

### `minibuffer-exit-hook`

[Variable]

This is a normal hook that is run whenever the minibuffer is exited. See Section 23.1 [Hooks], page 382.

### `minibuffer-help-form`

[Variable]

The current value of this variable is used to rebind `help-form` locally inside the minibuffer (see Section 24.5 [Help Functions], page 431).

### `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.

### `minibuffer-selected-window`

[Function]

This function returns the window which was selected when the minibuffer was entered. If selected window is not a minibuffer window, it returns `nil`.

**max-mini-window-height** [User Option]

This variable specifies the maximum height for resizing minibuffer windows. If a float, it specifies a fraction of the height of the frame. If an integer, it specifies a number of lines.

**minibuffer-message *string*** [Function]

This function displays *string* temporarily at the end of the minibuffer text, for two seconds, or until the next input event arrives, whichever comes first.

# 21 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.

## 21.1 Command Loop Overview

The first thing the command loop must do is read a key sequence, which is a sequence of events that translates into a command. It does this by calling the function `read-key-sequence`. Your Lisp code can also call this function (see Section 21.7.1 [Key Sequence Input], page 330). Lisp programs can also do input at a lower level with `read-event` (see Section 21.7.2 [Reading One Event], page 331) or discard pending input with `discard-input` (see Section 21.7.6 [Event Input Misc], page 335).

The key sequence is translated into a command through the currently active keymaps. See Section 22.10 [Key Lookup], page 358, 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 21.3 [Interactive Call], page 310).

To execute a command requires first reading the arguments for it. This is done by calling `command-execute` (see Section 21.3 [Interactive Call], page 310). For commands written in Lisp, the `interactive` specification says how to read the arguments. This may use the prefix argument (see Section 21.11 [Prefix Command Arguments], page 340) or may read with prompting in the minibuffer (see Chapter 20 [Minibuffers], page 278). For example, the command `find-file` has an `interactive` specification which says to read a file name using the minibuffer. The command's function body does not use the minibuffer; if you call this command from Lisp code as a function, you must supply the file name string as an ordinary Lisp function argument.

If the command is a string or vector (i.e., a keyboard macro) then `execute-kbd-macro` is used to execute it. You can call this function yourself (see Section 21.15 [Keyboard Macros], page 345).

To terminate the execution of a running command, type `C-g`. This character causes *quitting* (see Section 21.10 [Quitting], page 338).

### `pre-command-hook` [Variable]

The editor command loop runs this normal hook before each command. At that time, `this-command` contains the command that is about to run, and `last-command` describes the previous command. See Section 21.4 [Command Loop Info], page 312.

### `post-command-hook` [Variable]

The editor command loop runs this normal hook after each command (including commands terminated prematurely by quitting or by errors), and also when the command loop is first entered. At that time, `this-command` refers to the command that just ran, and `last-command` refers to the command before that.

Quitting is suppressed while running `pre-command-hook` and `post-command-hook`. If an error happens while executing one of these hooks, it terminates execution of the hook, and clears the hook variable to `nil` so as to prevent an infinite loop of errors.

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.

## 21.2 Defining Commands

A Lisp function becomes a command when its body contains, at top level, a form that calls the special form `interactive`. This form does nothing when actually executed, but its presence serves as a flag to indicate that interactive calling is permitted. Its argument controls the reading of arguments for an interactive call.

### 21.2.1 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 arg-descriptor` [Special Form]

This special form declares that the function in which it appears 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 has its effect because the command loop (actually, its subroutine `call-interactively`) scans through the function definition looking for it, before calling the function. Once the function is called, all its body forms including the `interactive` form are executed, but at this time `interactive` simply returns `nil` without even evaluating its argument.

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; then its contents should consist of a code character followed by a prompt (which some code characters use and some ignore). The prompt ends either with the end of the string or with a newline. Here is a simple example:

```
(interactive "bFrobinate buffer: ")
```

The code letter ‘`b`’ says to read the name of an existing buffer, with completion. The buffer name is the sole argument passed to the command. The rest of the string is a prompt.

If there is a newline character in the string, it terminates the prompt. If the string does not end there, then the rest of the string should contain another code character and prompt, specifying another argument. You can specify any number of arguments in this way.

The prompt string can use ‘`%`’ to include previous argument values (starting with the first argument) in the prompt. This is done using `format` (see Section 4.7 [Formatting

Strings], page 56). 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 the first character in the string is ‘\*’, then an error is signaled if the buffer is read-only.

If the first character in the string is ‘@’, 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.

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 20 [Minibuffers], page 278) or directly from the keyboard (see Section 21.7 [Reading Input], page 329).

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 21.3 [Interactive Call], page 310), 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.

### 21.2.2 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 20.6 [Completion], page 285). ? 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.
- ‘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 27 [Buffers], page 481). 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 30.1 [Point], page 559). No I/O.
- ‘D’ A directory name. The default is the current default directory of the current buffer, `default-directory` (see Section 25.8.4 [File Name Expansion], page 457). Existing, Completion, Default, Prompt.

'e'	The first or next mouse 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 21.6 [Input Events], page 315. No I/O.  You can use 'e' more than once in a single command's interactive specification. If the key sequence that invoked the command has <i>n</i> events that are lists, the <i>n</i> th 'e' provides the <i>n</i> 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 25.8 [File Names], page 453). The default directory is <b>default-directory</b> . 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 <b>nil</b> as the argument's value. No I/O.
'k'	A key sequence (see Section 22.1 [Key Sequences], page 347). 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 <b>describe-key</b> and <b>global-set-key</b> .
'K'	A key sequence, whose definition you intend to change. 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.
'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 <i>The GNU Emacs Manual</i> ). 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 21.11 [Prefix Command Arguments], page 340. 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. No I/O.

's'	Arbitrary text, read in the minibuffer and returned as a string (see Section 20.2 [Text from Minibuffer], page 279). Terminate the input with either <i>C-j</i> or RET. ( <i>C-q</i> may be used to include either of these characters in the input.) Prompt.
's'	An interned symbol whose name is read in the minibuffer. Any whitespace character terminates the input. (Use <i>C-q</i> to include whitespace in the string.) Other characters that normally terminate a symbol (e.g., parentheses and brackets) do not do so here. Prompt.
'U'	A key sequence or <code>nil</code> . 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 <code>nil</code> as the argument. No I/O.
'v'	A variable declared to be a user option (i.e., satisfying the predicate <code>user-variable-p</code> ). This reads the variable using <code>read-variable</code> . See [Definition of <code>read-variable</code> ], page 293. Existing, Completion, Prompt.
'x'	A Lisp object, specified with its read syntax, terminated with a <i>C-j</i> or RET. The object is not evaluated. See Section 20.3 [Object from Minibuffer], page 281. 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 <code>nil</code> . See Section 33.10 [Coding Systems], page 648. 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 <code>nil</code> as the argument value. Completion, Existing, Prompt.

### 21.2.3 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.
  (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
```

## 21.3 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.

**commandp** *object* &**optional** *for-call-interactively* [Function]  
 Returns `t` if *object* is suitable for calling interactively; that is, if *object* is a command. Otherwise, returns `nil`.

The interactively callable objects include strings and vectors (treated as keyboard macros), lambda expressions that contain a top-level call to `interactive`, byte-code function objects made from such lambda expressions, autoload objects that are declared as interactive (non-`nil` fourth argument to `autoload`), and some of the primitive functions.

A symbol satisfies `commandp` if its function definition satisfies `commandp`. Keys and keymaps are not commands. Rather, they are used to look up commands (see Chapter 22 [Keymaps], page 347).

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 24.2 [Accessing Documentation], page 426, for a realistic example of using `commandp`.

**call-interactively** *command* &**optional** *record-flag keys* [Function]  
 This function calls the interactively callable function *command*, reading arguments according to its interactive calling specifications. It returns whatever *command* returns. 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 21.14 [Command History], page 344.

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 314.

**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`, along with the optional *record-flag* and *keys*.

A symbol is handled by using its function definition 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 21.8 [Special Events], page 337).

**execute-extended-command** *prefix-argument* [Command]

This function reads a command name from the minibuffer using `completing-read` (see Section 20.6 [Completion], page 285). 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
```

**interactive-p** [Function]

This function returns *t* if the containing function (the one whose code includes the call to `interactive-p`) was called in direct response to user input. This means that it was called with the function `call-interactively`, and that a keyboard macro is not running, and that Emacs is not running in batch mode.

If the containing function was called by Lisp evaluation (or with `apply` or `funcall`), then it was not called interactively.

The most common use of `interactive-p` is for deciding whether to give the user additional visual feedback (such as by printing an informative message). For example:

```
; ; Here's the usual way to use interactive-p.
(defun foo ()
  (interactive)
  (when (interactive-p)
    (message "foo")))
⇒ foo

; ; This function is just to illustrate the behavior.
(defun bar ()
  (interactive)
  (setq foobar (list (foo) (interactive-p))))
⇒ bar

; ; Type M-x foo.
; ; Type M-x bar.
; ; This does not display a message.

foobar
⇒ (nil t)
```

If you want to test *only* whether the function was called using `call-interactively`, add an optional argument `print-message` which should be non-`nil` in an interactive call, and use the `interactive` spec to make sure it is non-`nil`. Here's an example:

```
(defun foo (&optional print-message)
  (interactive "p")
  (when print-message
    (message "foo")))
```

Defined in this way, the function does display the message when called from a keyboard macro. We use "`p`" because the numeric prefix argument is never `nil`.

**called-interactively-p** [Function]  
 This function returns `t` when the calling function was called using `call-interactively`.

When possible, instead of using this function, you should use the method in the example above; that method makes it possible for a caller to “pretend” that the function was called interactively.

## 21.4 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.

**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 29.2 [Multiple Displays], page 530.

**real-last-command**

[Variable]

This variable is set up by Emacs just like **last-command**, but never altered by Lisp programs.

**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 22.13 [Remapping Commands], page 364). 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.

**this-command-keys**

[Function]

This function returns a string or vector containing the key sequence that invoked the present command, plus any previous commands that generated the prefix argument

for this 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 21.7.1 [Key Sequence Input], page 330. The value is a string if all events in the sequence were characters that fit in a string. See Section 21.6 [Input Events], page 315.

```
(this-command-keys)
;; Now use C-u C-x C-e to evaluate that.
⇒ "U^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 21.6.14 [Strings of Events], page 328).

**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 39.12.2 [Recording Input], page 833) 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 20.7 [Yes-or-No Queries], page 296).

**last-command-event**

[Variable]

**last-command-char**

[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.

```
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`.

The alias `last-command-char` exists for compatibility with Emacs version 18.

**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 29.9 [Input Focus], page 543.

If the last event came from a keyboard macro, the value is `macro`.

## 21.5 Adjusting Point After Commands

It is not easy to display a value of point in the middle of a sequence of text that has the `display`, `composition` or `intangible` 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.

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.

## 21.6 Input Events

The Emacs command loop reads a sequence of *input events* that represent keyboard or mouse activity. The events for keyboard activity are characters or symbols; mouse 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 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. Instead, it distinguishes whether the symbol has actually been used in an event that has been read as input in the current Emacs session. If a symbol has not yet been so used, `eventp` returns `nil`.

### 21.6.1 Keyboard Events

There are two kinds of input you can get from the keyboard: ordinary keys, and function keys. Ordinary keys correspond to characters; the events they generate are represented in Lisp as characters. The event type of a character event is the character itself (an integer); see Section 21.6.12 [Classifying Events], page 324.

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).

shift	The $2^{25}$ bit in the character 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 characters.
	However, ASCII provides no way to distinguish <i>C-A</i> from <i>C-a</i> , so Emacs uses the $2^{25}$ bit in <i>C-A</i> and not in <i>C-a</i> .
hyper	The $2^{24}$ bit in the character code indicates a character typed with the hyper key held down.
super	The $2^{23}$ bit in the character code indicates a character typed with the super key held down.
alt	The $2^{22}$ bit in the character code indicates a character typed with the alt key held down. (On some terminals, the key labeled ALT is actually the meta key.)

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 21.6.12 [Classifying Events], page 324). When making key bindings, you can use the read syntax for characters with modifier bits ('*\C-*', '*\M-*', and so on). For making key bindings with `define-key`, you can use lists such as (`control hyper ?x`) to specify the characters (see Section 22.12 [Changing Key Bindings], page 361). The function `event-convert-list` converts such a list into an event type (see Section 21.6.12 [Classifying Events], page 324).

## 21.6.2 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 places the symbol `f1` in the input stream.

The event type of a function key event is the event symbol itself. See Section 21.6.12 [Classifying Events], page 324.

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 `function-key-map` (see Section 22.14 [Translation Keymaps], page 365) 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.

### 21.6.3 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 21.6.7 [Repeat Events], page 320). 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 21.2.2 [Interactive Codes], page 307.

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.

#### 21.6.4 Click Events

When the user presses a mouse button and releases it at the same location, that generates a *click* event. All mouse click 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. 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 the position where the mouse click occurred. The actual format of *position* depends on what part of a window was clicked on. The various formats are described below.

*click-count*

This is the number of rapid repeated presses so far of the same mouse button. See Section 21.6.7 [Repeat Events], page 320.

For mouse click events in the text area, mode line, header line, or in the marginal areas, *position* has this form:

```
(window pos-or-area (x . y) timestamp
      object text-pos (col . row)
      image (dx . dy) (width . height))
```

*window* This is the window in which the click occurred.

*pos-or-area*

This is the buffer position of the character clicked on in the text area, or if clicked outside the text area, it is the window area in which the click occurred. It is one of the symbols `mode-line`, `header-line`, `vertical-line`, `left-margin`, `right-margin`, `left-fringe`, or `right-fringe`.

*x, y*

These are the pixel-denominated coordinates of the click, relative to the top left corner of *window*, which is `(0 . 0)`. For the mode or header line, *y* does not have meaningful data. For the vertical line, *x* does not have meaningful data.

*timestamp*

This is the time at which the event occurred, in milliseconds.

*object*

This is the object on which the click occurred. It is either `nil` if there is no string property, or it has the form `(string . string-pos)` when there is a string-type text property at the click position.

*string*

This is the string on which the click occurred, including any properties.

<i>string-pos</i>	This is the position in the string on which the click occurred, relevant if properties at the click need to be looked up.
<i>text-pos</i>	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 other events, it is the current buffer position in the window.
<i>col, row</i>	These are the actual coordinates of the glyph under the <i>x, y</i> position, possibly padded with default character width glyphs if <i>x</i> is beyond the last glyph on the line.
<i>image</i>	This is the image object on which the click occurred. It is either <code>nil</code> if there is no image at the position clicked on, or it is an image object as returned by <code>find-image</code> if click was in an image.
<i>dx, dy</i>	These are the pixel-denominated coordinates of the click, relative to the top left corner of <i>object</i> , which is <code>(0 . 0)</code> . If <i>object</i> is <code>nil</code> , the coordinates are relative to the top left corner of the character glyph clicked on.

For mouse clicks on a scroll-bar, *position* has this form:

```
(window area (portion . whole) timestamp part)
```

<i>window</i>	This is the window whose scroll-bar was clicked on.
<i>area</i>	This is the scroll bar where the click occurred. It is one of the symbols <code>vertical-scroll-bar</code> or <code>horizontal-scroll-bar</code> .
<i>portion</i>	This is the distance of the click from the top or left end of the scroll bar.
<i>whole</i>	This is the length of the entire scroll bar.
<i>timestamp</i>	This is the time at which the event occurred, in milliseconds.
<i>part</i>	This is the part of the scroll-bar which was clicked on. It is one of the symbols <code>above-handle</code> , <code>handle</code> , <code>below-handle</code> , <code>up</code> , <code>down</code> , <code>top</code> , <code>bottom</code> , and <code>end-scroll</code> .

In one special case, *buffer-pos* 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 inserted into the input stream. See Section 21.7.1 [Key Sequence Input], page 330.

### 21.6.5 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 buffer-pos1 (x1 . y1) timestamp1)
  (window2 buffer-pos2 (x2 . y2) timestamp2)
  click-count)
```

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. Aside from that, the data have the same meanings as in a click event (see Section 21.6.4 [Click Events], page 318). You can access the second element of any mouse event in the same way, with no need to distinguish drag events from others.

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.

### 21.6.6 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.<sup>1</sup> These occur as soon as a button is pressed. They are represented by lists that look exactly like click events (see Section 21.6.4 [Click Events], page 318), 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 21.6.8 [Motion Events], page 321.

### 21.6.7 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.

---

<sup>1</sup> 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.

## 21.6.8 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 (window buffer-pos (x . y) timestamp))
```

The second element of the list describes the current position of the mouse, just as in a click event (see Section 21.6.4 [Click Events], page 318).

The special form `track-mouse` enables generation of motion events within its body. Outside of `track-mouse` forms, Emacs does not generate events for mere motion of the mouse, and these events do not appear. See Section 29.13 [Mouse Tracking], page 547.

### 21.6.9 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 29.9 [Input Focus], page 543.

Focus events are represented in Lisp as lists that look like this:

```
(switch-frame new-frame)
```

where `new-frame` is the frame switched to.

Most X window managers are set up so that just moving the mouse into a window is enough to set the focus there. Emacs appears to do this, because it changes the cursor to solid in the new frame. However, there is no need for the Lisp program to know about the focus change until some other kind of input arrives. So 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.

### 21.6.10 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.

```
(wheel-up position)
(wheel-down position)
```

These kinds of event are generated by moving a mouse wheel. Their usual meaning is a kind of scroll or zoom.

The element *position* is a list describing the position of the event, in the same format as used in a mouse-click event.

This kind of event is generated only on some kinds of systems. On some systems, `mouse-4` and `mouse-5` are used instead. For portable code, use the variables `mouse-wheel-up-event` and `mouse-wheel-down-event` defined in ‘`mwheel.el`’ to determine what event types to expect for the mouse wheel.

**(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, 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 621.

**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.

To catch a user signal, bind the corresponding event to an interactive command in the `special-event-map` (see Section 22.7 [Active Keymaps], page 353). The command is called with no arguments, and the specific signal event is available in `last-input-event`. For example:

```
(defun sigusr-handler ()
  (interactive)
  (message "Caught signal %S" last-input-event))

(define-key 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)
```

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.

### 21.6.11 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))
```

To handle a SIGUSR1 signal, define an interactive function, and bind it to the `signal usr1` event sequence:

```
(defun usr1-handler ()
  (interactive)
  (message "Got USR1 signal"))
(global-set-key [signal usr1] 'usr1-handler)
```

### 21.6.12 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`’.

#### `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.

**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
```

### 21.6.13 Accessing Events

This section describes convenient functions for accessing the data in a mouse button or motion event.

These two functions return the starting or ending position of a mouse-button event, as a list of this form:

```
(window pos-or-area (x . y) timestamp
      object text-pos (col . row)
      image (dx . dy) (width . height))
```

**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.

These functions take a position list as described above, and return various parts of it.

**posn-window** *position* [Function]

Return the window that *position* is in.

**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 these window-relative coordinates into frame-relative coordinates:

```
(defun frame-relative-coordinates (position)
  "Return frame-relative coordinates from POSITION."
  (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* [Function]

Return the row and column (in units of the frame's default character height and width) of *position*, as a cons cell (*col* . *row*). These are computed from the x and y values actually found in *position*.

**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 number in the window, and the actual character number in that row. It returns **nil** if *position* does not include actual positions values. You can use **posn-col-row** to get approximate values.

**posn-string** *position* [Function]

Return the string object in *position*, either **nil**, or a cons cell (*string* . *string-pos*).

**posn-image** *position* [Function]

Return the image object in *position*, either **nil**, or an image (*image* ...).

**posn-object** *position* [Function]

Return the image or string object in *position*, either **nil**, 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 in *position* as a cons cell (*dx* . *dy*). If the *position* is a buffer position, return the relative position in the character at that position.

**posn-object-width-height** *position* [Function]

Return the pixel width and height of the object in *position* as a cons cell (*width* . *height*). If the *position* is 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.

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 frame or window used. If *whole* is *nil*, the coordinates are relative to the window text area, otherwise they are relative to the entire window area including scroll bars, margins and fringes.

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.

### 21.6.14 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. Here is how to do that:

- Use vectors instead of strings for key sequences, when you plan to use them for anything other than as arguments to **lookup-key** and **define-key**. For example, you can use **read-key-sequence-vector** instead of **read-key-sequence**, and **this-command-keys-vector** instead of **this-command-keys**.

- Use vectors to write key sequence constants containing meta characters, even when passing them directly to `define-key`.
- When you have to look at the contents of a key sequence that might be a string, use `listify-key-sequence` (see Section 21.7.6 [Event Input Misc], page 335) first, to convert it to a list.

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.

## 21.7 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 38.8 [Temporary

Displays], page 752, and `sit-for` in Section 21.9 [Waiting], page 337. See Section 39.12 [Terminal Input], page 832, for functions and variables for controlling terminal input modes and debugging terminal input.

For higher-level input facilities, see Chapter 20 [Minibuffers], page 278.

### 21.7.1 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* *dont-downcase-last* [Function]  
*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 21.6.14 [Strings of Events], page 328). 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 22.14 [Translation Keymaps], page 365.

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 -----  

⇒ "^\u00c7F"
```

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 21.10 [Quitting], page 338.

**read-key-sequence-vector** *prompt* &**optional** *continue-echo* [Function]  
*dont-downcase-last* *switch-frame-ok* *command-loop*  
This is like `read-key-sequence` except that it always returns the key sequence as a vector, never as a string. See Section 21.6.14 [Strings of Events], page 328.

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. Note that `lookup-key` does not perform case conversion in this way.

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, 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: `header-line`, `horizontal-scroll-bar`, `menu-bar`, `mode-line`, `vertical-line`, and `vertical-scroll-bar`. 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.

## 21.7.2 Reading One Event

The lowest level functions for command input are those that read a single event.

None of the three functions below suppresses quitting.

**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. Events can come directly from the user or from a keyboard macro.

If the optional argument *prompt* is non-*nil*, it should be a string to display in the echo area as a prompt. Otherwise, **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 38.4 [The Echo Area], page 741.

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 value for *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 39.11 [Idle Timers], page 831)—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 24.5 [Help Functions], page 431. Certain other events, called *special events*, are also processed directly within **read-event** (see Section 21.8 [Special Events], page 337).

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 of command input. 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**.

In the first example, 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 of command input. If the user generates an event which is not a character, **read-char-exclusive** ignores it and reads another event, until it gets a character. The arguments work as in **read-event**.

**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.

### 21.7.3 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-\O, 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 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 93), 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 33.11 [Input Methods], page 659). Use `translation-table-for-input` (see Section 33.9 [Translation of Characters], page 647), if you want to translate characters after input methods operate.

`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.

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:

```
(keyboard-translate ?\C-x 'control-x)
(keyboard-translate ?\C-c 'control-c)
(keyboard-translate ?\C-v 'control-v)
(global-set-key [control-x] 'kill-region)
(global-set-key [control-c] 'kill-ring-save)
(global-set-key [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 22.14 [Translation Keymaps], page 365, for mechanisms that translate event sequences at the level of `read-key-sequence`.

#### 21.7.4 Invoking the Input Method

The event-reading functions invoke the current input method, if any (see Section 33.11 [Input Methods], page 659). 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 21.7.6 [Event Input Misc], page 335). 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.

### 21.7.5 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 21.10 [Quitting], page 338.

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
```

### 21.7.6 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 20.9 [Reading a Password], page 299).

`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 Section 21.6.14 [Strings of Events], page 328).

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.

`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`.

`unread-command-char` [Variable]

This variable holds a character to be read as command input. A value of -1 means “empty.”

This variable is mostly obsolete now that you can use `unread-command-events` instead; it exists only to support programs written for Emacs versions 18 and earlier.

`input-pending-p` [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.

`last-input-event` [Variable]

`last-input-char` [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
```

The alias `last-input-char` exists for compatibility with Emacs version 18.

`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)))
```

`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
```

## 21.8 Special Events

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.

Events that are handled in this way 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.

These 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`, 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 22.7 [Active Keymaps], page 353).

## 21.9 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 21.7.6 [Event Input Misc], page 335). Otherwise, the value is `nil`.

The argument `seconds` need not be an integer. If it is a floating point number, `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 38.2 [Forcing Redisplay], page 739.

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 39.16 [Batch Mode], page 836), `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 a floating point number, `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 39.5 [Time of Day], page 824, for functions to get the current time.

## 21.10 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 10.5.3 [Errors], page 127.)

You can specify a character other than `C-g` to use for quitting. See the function `set-input-mode` in Section 39.12 [Terminal Input], page 832.

## 21.11 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 21.2.1 [Using Interactive], page 305.) 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.

## 21.12 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 10.5.1 [Catch and Throw], page 125). If you throw a value other than `t`, then `recursive-edit` returns normally to the function that called it. The command `C-M-c` (`exit-recursive-edit`) does this. 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-J` (`abort-recursive-edit`).

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** [Function]

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 21.10 [Quitting], page 338.

#### **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.

## 21.13 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 39.1.2 [Init File], page 813) 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.

**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 39.1.2 [Init File], page 813) 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.

## 21.14 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 20.4 [Minibuffer History], page 282), 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 20.4 [Minibuffer History], page 282), 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.

## 21.15 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 13 [Macros], page 176).

**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 21.7.2 [Reading One Event], page 331, 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 29.2 [Multiple Displays], page 530.

**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 29.2 [Multiple Displays], page 530.

**kbd-macro-termination-hook**

[Variable]

This normal hook (see Appendix I [Standard Hooks], page 903) 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).

## 22 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*.

### 22.1 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, and mouse actions (see Section 21.6 [Input Events], page 315). 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, "`\Cx`" 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, "`\Cx1`" represents the key sequence `C-x 1`.

Key sequences containing function keys, mouse button 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 21.6 [Input Events], page 315. For example, the vector `[?\Cx ?1]` represents the key sequence `C-x 1`.

For examples of key sequences written in string and vector representations, section “Init Rebinding” in *The GNU Emacs Manual*.

#### `kbd keyseq-text` [Macro]

This macro converts the text `keyseq-text` (a string constant) into a key sequence (a string or vector constant). The contents of `keyseq-text` should describe the key sequence using almost the same syntax used in this manual. More precisely, it uses the same syntax that Edit Macro mode uses for editing keyboard macros (see section “Edit Keyboard Macro” in *The GNU Emacs Manual*); you must surround function key names with ‘`<...>`’.

```
(kbd "C-x") ⇒ "\Cx"
(kbd "C-x C-f") ⇒ "\Cx\C-f"
(kbd "C-x 4 C-f") ⇒ "\Cx4\C-f"
(kbd "X") ⇒ "X"
(kbd "RET") ⇒ "\^M"
(kbd "C-c SPC") ⇒ "\Cc "
(kbd "<f1> SPC") ⇒ [f1 32]
(kbd "C-M-<down>") ⇒ [C-M-down]
```

This macro is not meant for use with arguments that vary—only with string constants.

## 22.2 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 22.6 [Prefix Keys], page 352, 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 22.7 [Active Keymaps], page 353, for details.

## 22.3 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 21.6.12 [Classifying Events], page 324. In this kind of binding, `binding` is a command.

`(type item-name [cache] . binding)`

This specifies a binding which is also a simple menu item that displays as *item-name* in the menu. *cache*, if present, caches certain information for display in the menu. See Section 22.17.1.1 [Simple Menu Items], page 371.

`(type item-name help-string [cache] . 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 22.17.1.2 [Extended Menu Items], page 372.

`(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 13): element *n* is the binding for the character with code *n*. 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*.

`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 22.17.1 [Defining Menus], page 370.

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 22.6 [Prefix Keys], page 352).

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 and TAB, plus `C-c C-l`, `M-C-q`, and `M-C-x`.

```
lisp-mode-map
⇒
(keymap
 (3 keymap
   ;; C-c C-z
 (26 . run-lisp))
```

```
(27 keymap
  ;; M-C-x, treated as ESC C-x
  (24 . lisp-send-defun)
  keymap
  ;; M-C-q, treated as ESC C-q
  (17 . indent-sexp))
;; This part is inherited from lisp-mode-shared-map.
keymap
;; DEL
(127 . backward-delete-char-untabify)
(27 keymap
  ;; M-C-q, treated as ESC C-q
  (17 . indent-sexp))
(9 . lisp-indent-line))
```

**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
```

## 22.4 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 22.17.1 [Defining Menus], page 370). 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 93) 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 #^[[t nil nil nil ... nil nil keymap])
```

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.

`copy-keymap keymap` [Function]

This function returns a copy of *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
```

## 22.5 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 `define-key` 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 `define-key` 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.

## 22.6 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.
- `vc-prefix-map` is the global keymap used for the `C-x v` prefix key.
- `facemenu-keymap` is the global keymap used for the `M-o` prefix key.
- The other Emacs prefix keys are `M-g`, `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 22.7 [Active Keymaps], page 353.

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*. The key sequence *C-p 6* is not found in any active keymap.

```
(use-local-map (make-sparse-keymap))
  ⇒ nil
(local-set-key "\C-p" ctl-x-map)
  ⇒ nil
(key-binding "\C-p\C-f")
  ⇒ find-file

(key-binding "\C-p6")
  ⇒ 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 22.17.1 [Defining Menus], page 370).

## 22.7 Active Keymaps

Emacs normally contains many keymaps; at any given time, just a few of them are *active*, meaning that they participate in the interpretation of user input. All the active keymaps are used together to determine what command to execute when a key is entered.

Normally the active keymaps are the `keymap` property keymap, the keymaps of any enabled minor modes, the current buffer's local keymap, and the global keymap, in that order. Emacs searches for each input key sequence in all these keymaps. See Section 22.8 [Searching Keymaps], page 355, for more details of this procedure.

When the key sequence starts with a mouse event (optionally preceded by a symbolic prefix), the active keymaps are determined based on the position in that event. If the event happened on a string embedded with a `display`, `before-string`, or `after-string` property (see Section 32.19.4 [Special Properties], page 620), the non-*nil* map properties of the string override those of the buffer.

The `global keymap` holds the bindings of keys that are defined regardless of the current buffer, such as *C-f*. The variable `global-map` holds this keymap, which is always active.

Each buffer may have another keymap, its *local keymap*, which may contain new or overriding definitions for keys. The current buffer's local keymap is always active except when `overriding-local-map` overrides it. The `local-map` text or overlay property can specify an alternative local keymap for certain parts of the buffer; see Section 32.19.4 [Special Properties], page 620.

Each minor mode can have a keymap; if it does, the keymap is active when the minor mode is enabled. Modes for emulation can specify additional active keymaps through the variable `emulation-mode-map-alists`.

The highest precedence normal keymap comes from the `keymap` text or overlay property. If that is `non-nil`, it is the first keymap to be processed, in normal circumstances.

However, there are also special ways for programs to substitute other keymaps for some of those. The variable `overriding-local-map`, if `non-nil`, specifies a keymap that replaces all the usual active keymaps except the global keymap. Another way to do this is with `overriding-terminal-local-map`; it operates on a per-terminal basis. These variables are documented below.

Since every buffer that uses the same major mode normally uses the same local keymap, you can think of the keymap as local to the mode. A change to the local keymap of a buffer (using `local-set-key`, for example) is seen also in the other buffers that share that keymap.

The local keymaps that are used for Lisp mode and some other major modes exist even if they have not yet been used. These local keymaps are the values of variables such as `lisp-mode-map`. For most major modes, which are less frequently used, the local keymap is constructed only when the mode is used for the first time in a session.

The minibuffer has local keymaps, too; they contain various completion and exit commands. See Section 20.1 [Intro to Minibuffers], page 278.

Emacs has other keymaps that are used in a different way—translating events within `read-key-sequence`. See Section 22.14 [Translation Keymaps], page 365.

See Appendix H [Standard Keymaps], page 899, for a list of standard keymaps.

**current-active-maps** &**optional** *olp* [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.

**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 22.11 [Functions for Key Lookup], page 360).

When commands are remapped (see Section 22.13 [Remapping Commands], page 364), `key-binding` normally processes command remappings so as to return the remapped command that will actually be executed. However, if *no-remap* is `non-nil`, `key-binding` ignores remappings and returns the binding directly specified for *key*.

If *key* starts with a mouse event (perhaps following a prefix event), the maps to be consulted are determined based on the event's position. Otherwise, they are determined based on the value of *point*. However, you can override either of them by specifying *position*. If *position* is non-nil, it should be either a buffer position or an event position like the value of `event-start`. Then the maps consulted are determined based on *position*.

An error is signaled if *key* is not a string or a vector.

```
(key-binding "\C-x\C-f")
⇒ find-file
```

## 22.8 Searching the Active Keymaps

After translation of event subsequences (see Section 22.14 [Translation Keymaps], page 365) Emacs looks for them in the active keymaps. Here is a pseudo-Lisp description of the order and conditions for searching them:

```
(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-text-property (point) 'local-map)
                (find-in (get-char-property (point) 'local-map))
              (find-in (current-local-map))))))
      (find-in (current-global-map)))
```

The `find-in` and `find-in-any` are pseudo functions that search in one keymap and in an alist of keymaps, respectively. (Searching a single keymap for a binding is called *key lookup*; see Section 22.10 [Key Lookup], page 358.) If the key sequence starts with a mouse event, or a symbolic prefix event followed by a mouse event, that event's position is used instead of *point* and the current buffer. Mouse events on an embedded string use non-nil text properties from that string instead of the buffer.

1. The function finally found may be remapped (see Section 22.13 [Remapping Commands], page 364).
2. Characters that are bound to `self-insert-command` are translated according to `translation-table-for-input` before insertion.
3. `current-active-maps` returns a list of the currently active keymaps at point.
4. When a match is found (see Section 22.10 [Key Lookup], page 358), if the binding in the keymap is a function, the search is over. However if the keymap entry 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.

## 22.9 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.

```
(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-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 23.3.2 [Keymaps and Minor Modes], page 399.

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 23.3.2 [Keymaps and Minor Modes], page 399, for more information about minor modes. See also `minor-mode-key-binding` (see Section 22.11 [Functions for Key Lookup], page 360).

#### `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 29.2 [Multiple Displays], page 530. 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 "unreadable" 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 21.8 [Special Events], page 337.

**emulation-mode-map-alists**

[Variable]

This variable holds a list of keymap alists to use for emulations 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`.

## 22.10 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 21.6 [Input Events], page 315. 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 `lookup-key` 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:

<code>nil</code>	<code>nil</code> 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 <code>nil</code> for that event type.
<code>command</code>	The events used so far in the lookup form a complete key, and <code>command</code> is its binding. See Section 12.1 [What Is a Function], page 160.
<code>array</code>	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 21.15 [Keyboard Macros], page 345, for more information.
<code>keymap</code>	The events used so far in the lookup form a prefix key. The next event of the key sequence is looked up in <code>keymap</code> .
<code>list</code>	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 21.2 [Defining Commands], page 305.
- If the CAR of *list* is a keymap and the CDR is an event type, then this is an *indirect entry*:

`(othermap . othertype)`

When key lookup encounters an indirect entry, it looks up instead the binding of *othertype* in *othermap* and uses that.

This feature permits you to define one key as an alias for another key. For example, an entry whose CAR is the keymap called `esc-map` and whose CDR is 32 (the code for SPC) means, “Use the global binding of *Meta-SPC*, whatever that may be.”

**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. A list is allowed if it is a keymap or a command, but indirect entries are not understood when found via symbols.

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 21.3 [Interactive Call], page 310).

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 an indirection or `nil`. Here is an example of a sparse keymap with two characters bound to commands and one bound to another keymap. This map is the normal value of `emacs-lisp-mode-map`. Note that 9 is the code for TAB, 127 for DEL, 27 for ESC, 17 for *C-q* and 24 for *C-x*.

```
(keymap (9 . lisp-indent-line)
        (127 . backward-delete-char-untabify)
        (27 keymap (17 . indent-sexp) (24 . eval-defun)))
```

## 22.11 Functions for Key Lookup

Here are the functions and variables pertaining to key lookup.

**lookup-key** *keymap key &optional accept-defaults* [Function]

This function returns the definition of *key* in *keymap*. All the other functions described in this chapter that look up keys use **lookup-key**. Here are examples:

```
(lookup-key (current-global-map) "\C-x\C-f")
⇒ find-file
(lookup-key (current-global-map) (kbd "C-x C-f"))
⇒ find-file
(lookup-key (current-global-map) "\C-x\C-f12345")
⇒ 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 **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. (To do this, supply *t* as an element of *key*; see Section 22.3 [Format of Keymaps], page 348.)

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.

```
(lookup-key (current-global-map) "\M-f")
⇒ forward-word
(lookup-key (current-global-map) "\ef")
⇒ forward-word
```

Unlike **read-key-sequence**, this function does not modify the specified events in ways that discard information (see Section 21.7.1 [Key Sequence Input], page 330). In particular, it does not convert letters to lower case and it does not change drag events to clicks.

**undefined** [Command]

Used in keymaps to undefine keys. It calls **ding**, but does not cause an error.

**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).

**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 **lookup-key** (above).

**meta-prefix-char** [Variable]

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 22.6 [Prefix Keys], page 352). 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*.

## 22.12 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 `global-set-key` and `local-set-key` functions are convenient interfaces for these operations (see Section 22.15 [Key Binding Commands], page 367). You can also use `define-key`, a more general function; then you must specify explicitly 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 859).

In writing the key sequence to rebind, it is good to use the special escape sequences for control and meta characters (see Section 2.3.8 [String Type], page 18). The syntax '`\C-`' means that the following character is a control character and '`\M-`' means that the following character is a meta character. Thus, the string "`\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`. You can also use this escape syntax in vectors, as well as others that aren't allowed in strings; one example is '`[?\C-\H-x home]`'. See Section 2.3.3 [Character Type], page 10.

The key definition and lookup functions accept an alternate syntax for event types in a key sequence that is a vector: you can use a list containing modifier names plus one base event (a character or function key name). For example, `(control ?a)` is equivalent to `?C-a` and `(hyper control left)` is equivalent to `C-H-left`. One advantage of such lists is that the precise numeric codes for the modifier bits don't appear in compiled files.

The functions below signal an error if `keymap` is not a keymap, or if `key` is not a string or vector representing a key sequence. You can use event types (symbols) as shorthand for events that are lists. The `kbd` macro (see Section 22.1 [Key Sequences], page 347) is a convenient way to specify the key sequence.

**define-key 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 22.10 [Key Lookup], page 358.) The value returned by `define-key` 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 `define-key` 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 22.17 [Menu Keymaps], page 370).

This example creates a sparse keymap and makes a number of bindings in it:

```
(setq map (make-sparse-keymap))
  ⇒ (keymap)
(define-key map "\C-f" 'forward-char)
  ⇒ forward-char
```

```

map
⇒ (keymap (6 . forward-char))

;; Build sparse submap for C-x and bind f in that.
(define-key map (kbd "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.
(define-key map (kbd "C-p") ctl-x-map)
;; ctl-x-map
⇒ [nil ... find-file ... backward-kill-sentence]

;; Bind C-f to foo in the ctl-x-map.
(define-key map (kbd "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.

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 to remap one command into another (see Section 22.13 [Remapping Commands], page 364).

**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 rebinding 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 '(keymap
  (?1 . olddef-1)
  (?2 . olddef-2)
  (?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 22.13 [Remapping Commands], page 364). 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 27.7 [Read Only Buffers], page 489).

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.

Most often, **suppress-keymap** is used to initialize local keymaps of modes such as Rmail and Dired where insertion of text is not desirable and the buffer is read-only. Here is an example taken from the file ‘**emacs/lisp/dired.el**’, showing how the local keymap for Dired mode is set up:

```
(setq dired-mode-map (make-keymap))
(suppress-keymap dired-mode-map)
(define-key dired-mode-map "r" 'dired-rename-file)
(define-key dired-mode-map "\C-d" 'dired-flag-file-deleted)
(define-key dired-mode-map "d" 'dired-flag-file-deleted)
(define-key dired-mode-map "v" 'dired-view-file)
(define-key dired-mode-map "e" 'dired-find-file)
(define-key dired-mode-map "f" 'dired-find-file)
...
```

## 22.13 Remapping Commands

A special kind of key binding, using a special “key sequence” which includes a command name, has the effect of *remapping* that command into another. Here’s how it works. You make a key binding for a key sequence that starts with the dummy event **remap**, followed by the command name you want to remap. Specify the remapped definition as the definition in this binding. The remapped definition is usually a command name, but it can be any valid definition for a key binding.

Here's an example. Suppose that My mode uses special commands `my-kill-line` and `my-kill-word`, which should be invoked instead of `kill-line` and `kill-word`. It can establish this by making these two command-remapping bindings in its keymap:

```
(define-key my-mode-map [remap kill-line] 'my-kill-line)
(define-key my-mode-map [remap kill-word] 'my-kill-word)
```

Whenever `my-mode-map` is an active keymap, if the user types `C-k`, Emacs will find the standard global binding of `kill-line` (assuming nobody has changed it). But `my-mode-map` remaps `kill-line` to `my-kill-line`, so instead of running `kill-line`, Emacs runs `my-kill-line`.

Remapping only works through a single level. In other words,

```
(define-key my-mode-map [remap kill-line] 'my-kill-line)
(define-key my-mode-map [remap my-kill-line] 'my-other-kill-line)
```

does not have the effect of remapping `kill-line` into `my-other-kill-line`. If an ordinary key binding specifies `kill-line`, this keymap will remap it to `my-kill-line`; if an ordinary binding specifies `my-kill-line`, this keymap will remap it to `my-other-kill-line`.

#### `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`.

## 22.14 Keymaps for Translating Sequences of Events

This section describes keymaps that are used during reading a key sequence, to translate certain event sequences into others. `read-key-sequence` checks every subsequence of the key sequence being read, as it is read, against `function-key-map` and then against `key-translation-map`.

#### `function-key-map` [Variable]

This variable holds a keymap that describes the character sequences sent by function keys on an ordinary character terminal. This keymap has the same structure as other keymaps, but is used differently: it specifies translations to make while reading key sequences, rather than bindings for key sequences.

If `function-key-map` "binds" a key sequence *k* to a vector *v*, then when *k* appears as a subsequence *anywhere* in a key sequence, it is replaced with the events in *v*.

For example, VT100 terminals send `ESC O P` when the keypad PF1 key is pressed. Therefore, we want Emacs to translate that sequence of events into the single event `pf1`. We accomplish this by "binding" `ESC O P` to `[pf1]` in `function-key-map`, when using a VT100.

Thus, typing `C-c PF1` sends the character sequence `C-c ESC O P`; later the function `read-key-sequence` translates this back into `C-c PF1`, which it returns as the vector `[?C-c pf1]`.

Entries in `function-key-map` are ignored if they conflict with bindings made in the minor mode, local, or global keymaps. The intent is that the character sequences

that function keys send should not have command bindings in their own right—but if they do, the ordinary bindings take priority.

The value of `function-key-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 `function-key-map` beyond those that can be deduced from Termcap and Terminfo. See Section 39.1.3 [Terminal-Specific], page 814.

## key-translation-map

[Variable]

This variable is another keymap used just like `function-key-map` to translate input events into other events. It differs from `function-key-map` in two ways:

- `key-translation-map` goes to work after `function-key-map` is finished; it receives the results of translation by `function-key-map`.
  - Non-prefix bindings in `key-translation-map` override actual key bindings. For example, if `C-x f` has a non-prefix binding in `key-translation-map`, that translation takes effect even though `C-x f` also has a key binding in the global map.

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 `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 `function-key-map` or `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:

```
(if (symbolp e)
    symbol
  (cons symbol (cdr e)))))

(define-key function-key-map "\C-ch" 'hyperify)
```

If you have enabled keyboard character set decoding using `set-keyboard-coding-system`, decoding is done after the translations listed above. See Section 33.10.8 [Terminal I/O Encoding], page 657. However, in future Emacs versions, character set decoding may be done at an earlier stage.

## 22.15 Commands for Binding Keys

This section describes some convenient interactive interfaces for changing key bindings. They work by calling `define-key`.

People often use `global-set-key` in their init files (see Section 39.1.2 [Init File], page 813) for simple customization. For example,

```
(global-set-key (kbd "C-x C-\\")) 'next-line)
```

or

```
(global-set-key [?\C-x ?\C-\\")) 'next-line)
```

or

```
(global-set-key [(control ?x) (control ?\\))] 'next-line)
```

redefines *C-x C-\* to move down a line.

```
(global-set-key [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 15.4 [Loading Non-ASCII], page 205), you must type the keys as multibyte too. For instance, if you use this:

```
(global-set-key "ö" 'my-function) ; bind o-umlaut
```

or

```
(global-set-key ?ö 'my-function) ; bind o-umlaut
```

and your language environment is multibyte Latin-1, these commands actually bind the multibyte character with code 2294, not the unibyte Latin-1 character with code 246 (*M-v*). In order to use this binding, you need to enter the multibyte Latin-1 character as keyboard input. One way to do this is by using an appropriate input method (see section “Input Methods” in *The GNU Emacs Manual*).

If you want to use a unibyte character in the key binding, you can construct the key sequence string using `multibyte-char-to-unibyte` or `string-make-unibyte` (see Section 33.2 [Converting Representations], page 641).

**global-set-key** *key binding* [Command]

This function sets the binding of *key* in the current global map to *binding*.

```
(global-set-key key binding)
≡
(define-key (current-global-map) key binding)
```

**global-unset-key** *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:

```
(global-unset-key "\C-1")
  ⇒ nil
(global-set-key "\C-1\C-1" 'redraw-display)
  ⇒ nil
```

This function is implemented simply using **define-key**:

```
(global-unset-key key)
≡
(define-key (current-global-map) key nil)
```

**local-set-key** *key binding*

[Command]

This function sets the binding of *key* in the current local keymap to *binding*.

```
(local-set-key key binding)
≡
(define-key (current-local-map) key binding)
```

**local-unset-key** *key*

[Command]

This function removes the binding of *key* from the current local map.

```
(local-unset-key key)
≡
(define-key (current-local-map) key nil)
```

## 22.16 Scanning Keymaps

This section describes functions used to scan all the current keymaps for the sake of printing help information.

**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`

[Function]

*no-remap*

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 (*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 follow indirect keymap bindings. This makes it possible to search for an indirect definition itself.

When command remapping is in effect (see Section 22.13 [Remapping Commands], page 364), `where-is-internal` figures out when a command will be run due to remapping and reports keys accordingly. It also returns `nil` if `command` won't really be run because it has been remapped to some other command. However, if `no-remap` is non-`nil`. `where-is-internal` ignores remappings.

```
(where-is-internal 'describe-function)
⇒ ([8 102] [f1 102] [help 102]
    [menu-bar help-menu describe describe-function])
```

**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`.

The listing describes meta characters as ESC followed by the corresponding non-meta character.

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.

## 22.17 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.

### 22.17.1 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 22.3 [Format of Keymaps], page 348.) 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.<sup>1</sup> 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 22.4 [Creating Keymaps], page 350), or `define-prefix-command` (see [Definition of define-prefix-

---

<sup>1</sup> It is required for menus which do not use a toolkit, e.g. under MS-DOS.

command], page 353). 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 `define-key-after` (see Section 22.17.7 [Modifying Menus], page 381).

### 22.17.1.1 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 it is not generally possible to display non-ASCII text in menus. It will work for keyboard menus and will work to a large extent when Emacs is built with the Gtk+ toolkit.<sup>2</sup>

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 624).

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 23.4 [Mode Line Format], page 402).

---

<sup>2</sup> In this case, the text is first encoded using the utf-8 coding system and then rendered by the toolkit as it sees fit.

You've probably noticed that menu items show the equivalent keyboard key sequence (if any) to invoke the same command. To save time on recalculation, menu display caches this information in a sublist in the binding, like this:

```
(item-string [help] (key-binding-data) . real-binding)
```

Don't put these sublists in the menu item yourself; menu display calculates them automatically. Don't mention keyboard equivalents in the item strings themselves, since that is redundant.

### 22.17.1.2 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 22.17.1.3 [Menu Separators], page 374.

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*, is the command to execute. The tail of the list, *item-property-list*, has the form of a property list which contains other information.

When an equivalent keyboard key binding is cached, the extended menu item binding looks like this:

```
(menu-item item-name real-binding (key-binding-data)  
  . item-property-list)
```

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 624). 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 is likely to be bound to the same command invoked by this menu item. If you specify the right key sequence, that makes preparing the menu for display run much faster.

If you specify the wrong key sequence, it has no effect; before Emacs displays *key-sequence* in the menu, it verifies that *key-sequence* is really equivalent to this menu item.

#### **:key-sequence nil**

This property indicates that there is normally no key binding which is equivalent to this menu item. Using this property saves time in preparing the menu for display, because Emacs does not need to search the keymaps for a keyboard equivalent for this menu item.

However, if the user has rebound this item’s definition to a key sequence, Emacs ignores the **:keys** property and finds the keyboard equivalent anyway.

#### **: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*.

#### **: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.

### 22.17.1.3 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 for 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`".

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.

#### 22.17.1.4 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 `toggle-read-only` and gives them different enable conditions:

```
(defalias 'make-read-only 'toggle-read-only)
(put 'make-read-only 'menu-enable '(not buffer-read-only))
(defalias 'make-writable 'toggle-read-only)
(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 `toggle-read-only`.

#### 22.17.2 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 29.15 [Pop-Up Menus], page 548.)

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.

A single keymap can appear as multiple menu panes, if you explicitly arrange for this. The way to do this is to make a keymap for each pane, then create a binding for each of those maps in the main keymap of the menu. Give each of these bindings an item string that starts with '`@`'. The rest of the item string becomes the name of the pane. See the file '`lisp/mouse.el`' for an example of this. Any ordinary bindings with '`@`'-less item strings are grouped into one pane, which appears along with the other panes explicitly created for the submaps.

X toolkit menus don't have panes; instead, they can have submenus. Every nested keymap becomes a submenu, whether the item string starts with ‘`C-`’ or not. In a toolkit version of Emacs, the only thing special about ‘`C-`’ at the beginning of an item string is that the ‘`C-`’ doesn't appear in the menu item.

Multiple keymaps that define the same menu prefix key produce separate panes or separate submenus.

### 22.17.3 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.

This way of using menus in an Emacs-like editor was inspired by the Hierarkey system.

`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.

### 22.17.4 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 22.17.1.2 [Extended Menu Items], page 372). 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" tags-loop-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)
```

## 22.17.5 The Menu Bar

Most window systems allow each frame to have a *menu bar*—a permanently displayed menu stretching horizontally across the top of the frame. The items of the menu bar are the 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 fake 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 22.7 [Active Keymaps], page 353.

In order for a frame to display a menu bar, its `menu-bar-lines` parameter must be greater than zero. Emacs uses just one line for the menu bar itself; if you specify more than one line, the other lines serve to separate the menu bar from the windows in the frame. We recommend 1 or 2 as the value of `menu-bar-lines`. See Section 29.3.3.4 [Layout Parameters], page 534.

Here’s an example of setting up a menu bar item:

```
(modify-frame-parameters (selected-frame)
  '((menu-bar-lines . 2)))

;; 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)
```

`edit` is the fake function key 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 submenus 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.

### 22.17.6 Tool bars

A *tool bar* is a row of icons at the top of a frame, that execute commands when you click on them—in effect, a kind of graphical menu bar.

The frame parameter `tool-bar-lines` (X resource ‘`toolBar`’) 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 of `auto-resize-tool-bars` 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`.

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 22.17.1.2 [Extended Menu Items], page 372), 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 624).

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 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.

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 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 23.2.2 [Major Mode Conventions], page 385). 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:

```
(global-set-key [tool-bar]
  '(menu-item "tool bar" ignore
    :filter (lambda (ignore) tool-bar-map)))
```

Thus the tool bar map is derived dynamically from the value of variable `tool-bar-map` and you should normally adjust the default (global) tool bar by changing that map. Major modes may replace the global bar completely by making `tool-bar-map` buffer-local and set to a keymap containing only the desired items. Info mode provides an example.

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-bar** [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**.

**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 value 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 21.6.2 [Function Keys], page 316, for more information about how to add modifiers to function keys.

### 22.17.7 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 `define-key-after`:

**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.

Here is an example:

```
(define-key-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`:

```
(define-key-after
  (lookup-key shell-mode-map [menu-bar signals])
  [work] ('("Work" . work-command) 'break))
```

## 23 Major and Minor Modes

A *mode* is a set of definitions that customize Emacs and can be turned on and off while you edit. There are two varieties of modes: *major modes*, which are mutually exclusive and used for editing particular kinds of text, and *minor modes*, which provide features that users can enable individually.

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 22 [Keymaps], page 347, and Chapter 35 [Syntax Tables], page 684.

### 23.1 Hooks

A *hook* is a variable where you can store a function or functions 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 39.1.2 [Init File], page 813), but Lisp programs can set them also. See Appendix I [Standard Hooks], page 903, for a list of 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 function is supposed to run a normal hook called the *mode hook* as the 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 39.2.2 [Suspending Emacs], page 817).

The recommended way to add a hook function to a normal hook is by calling **add-hook** (see below). The hook functions may be any of the valid kinds of functions that **funcall** accepts (see Section 12.1 [What Is a Function], page 160). 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**.

If the hook variable’s name does not end with ‘-hook’, that indicates it is probably an *abnormal hook*. That means the hook functions are called with arguments, or their return values are used in some way. The hook’s documentation says how the functions are called. You can use **add-hook** to add a function to an abnormal hook, but you must write the function to follow the hook’s calling convention.

By convention, abnormal hook names end in ‘-functions’ or ‘-hooks’. If the variable’s name ends in ‘-function’, then its value is just a single function, not a list of functions.

Here’s an example that uses a mode hook to turn on Auto Fill mode when in Lisp Interaction mode:

```
(add-hook 'lisp-interaction-mode-hook 'turn-on-auto-fill)
```

At the appropriate time, Emacs uses the **run-hooks** function to run particular 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.

**run-hook-with-args** hook &rest args [Function]

This function is the way to run an abnormal hook and always call all of the hook functions. It calls each of the hook functions one by one, passing each of them the arguments *args*.

**run-hook-with-args-until-failure** hook &rest args [Function]

This function is the way to run an abnormal hook until one of the hook functions fails. It calls each of the hook functions, passing each of them the arguments *args*, until some hook function returns *nil*. It then stops and returns *nil*. If none of the hook functions return *nil*, it returns a non-*nil* value.

**run-hook-with-args-until-success** hook &rest args [Function]

This function is the way to run an abnormal hook until a hook function succeeds. It calls each of the hook functions, passing each of them the arguments *args*, until some hook function returns non-*nil*. Then it stops, and returns whatever was returned by the last hook function that was called. If all hook functions return *nil*, it returns *nil* as well.

**add-hook** hook function &optional append 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.

It is best to design your hook functions 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 will be executed first (barring another **add-hook** call). If the optional argument *append* is non-*nil*, the new hook function goes at the end of the hook list and will be executed last.

**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. If needed, 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.

## 23.2 Major Modes

Major modes specialize Emacs for editing particular kinds of text. Each buffer has only one major mode at a time. For each major mode there is a function to switch to that mode in the current buffer; its name should end in ‘`-mode`’. These functions work by setting buffer-local variable bindings and other data associated with the buffer, such as a local keymap. The effect lasts until you switch to another major mode in the same buffer.

### 23.2.1 Major Mode Basics

The least specialized major mode is called *Fundamental mode*. This mode has no mode-specific definitions or variable settings, so each Emacs command behaves in its default manner, and each option is in its default state. All other major modes redefine various keys and options. For example, Lisp Interaction mode provides special key bindings for `C-j` (`eval-print-last-sexp`), TAB (`lisp-indent-line`), and other keys.

When you need to write several editing commands to help you perform a specialized editing task, creating a new major mode is usually a good idea. In practice, writing a major mode is easy (in contrast to writing a minor mode, which is often difficult).

If the new mode is similar to an old one, it is often unwise to modify the old one to serve two purposes, since it may become harder to use and maintain. Instead, copy and rename an existing major mode definition and alter the copy—or use `define-derived-mode` to define a *derived mode* (see Section 23.2.5 [Derived Modes], page 391). For example, Rmail Edit mode is a major mode that is very similar to Text mode except that it provides two additional commands. Its definition is distinct from that of Text mode, but uses that of Text mode.

Even if the new mode is not an obvious derivative of any other mode, it is convenient to use `define-derived-mode` with a `nil` parent argument, since it automatically enforces the most important coding conventions for you.

For a very simple programming language major mode that handles comments and fontification, you can use `define-generic-mode`. See Section 23.2.6 [Generic Modes], page 392.

Rmail Edit mode offers an example of changing the major mode temporarily for a buffer, so it can be edited in a different way (with ordinary Emacs commands rather than Rmail commands). In such cases, the temporary major mode usually provides a command to switch back to the buffer’s usual mode (Rmail mode, in this case). You might be tempted to present the temporary redefinitions inside a recursive edit and restore the usual ones when the user exits; but this is a bad idea because it constrains the user’s options when it is done in more than one buffer: recursive edits must be exited most-recently-entered

first. Using an alternative major mode avoids this limitation. See Section 21.12 [Recursive Editing], page 342.

The standard GNU Emacs Lisp library directory tree contains the code for several major modes, in files such as ‘`text-mode.el`’, ‘`texinfo.el`’, ‘`lisp-mode.el`’, ‘`c-mode.el`’, and ‘`rmail.el`’. They are found in various subdirectories of the ‘`lisp`’ directory. You can study these libraries to see how modes are written. Text mode is perhaps the simplest major mode aside from Fundamental mode. Rmail mode is a complicated and specialized mode.

### 23.2.2 Major Mode Conventions

The code for existing major modes follows various coding conventions, including conventions for local keymap and syntax table initialization, global names, and hooks. Please follow these conventions when you define a new major mode. (Fundamental mode is an exception to many of these conventions, because its definition is to present the global state of Emacs.)

This list of conventions is only partial, because each major mode should aim for consistency in general with other Emacs major modes. 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 command whose name ends in ‘`-mode`’, with no arguments, that switches to the new mode in the current buffer. This command should set up the keymap, syntax table, and buffer-local variables in an existing buffer, without changing the buffer’s contents.
- Write a documentation string for this command that describes the special commands available in this mode. `C-h m (describe-mode)` in your mode will display this string.

The documentation string may include the special documentation substrings, ‘`\[command]`’, ‘`\{keymap}`’, and ‘`\<keymap>`’, which enable the documentation to adapt automatically to the user’s own key bindings. See Section 24.3 [Keys in Documentation], page 428.

- 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 11.10.2 [Creating Buffer-Local], page 149.
- 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, as a string. This string appears in the mode line.
- 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 856.
- 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.

- 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 22.7 [Active Keymaps], page 353, 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 11.6 [Tips for Defining], page 141, 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 rebinding 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 rebinding 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.

It is also legitimate for a major mode to rebinding a standard key sequence whose standard meaning is rarely useful in that mode. For instance, minibuffer modes rebinding `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 35 [Syntax Tables], page 684.
- 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 36.3 [Defining Abbrevs], page 700.

- 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 23.6 [Font Lock Mode], page 412).
- 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 23.5 [Imenu], page 410).
- The mode can specify a local value for `eldoc-documentation-function` to tell ElDoc mode how to handle this mode.
- Use `defvar` or `defcustom` to set mode-related variables, so that they are not reinitialized if they already have a value. (Such reinitialization could discard customizations made by the user.)
- 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 11.10 [Buffer-Local Variables], page 147.

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 mode hook, and then runs the normal hook `after-change-major-mode-hook`. See Section 23.2.7 [Mode Hooks], page 393.
- 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 `define-derived-mode`, 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 23.2.5 [Derived Modes], page 391, and Section 23.2.7 [Mode Hooks], page 393.
- 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 11.10.2 [Creating Buffer-Local], page 149).
- If this mode is appropriate only for specially-prepared text, 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 inherit Funny mode, in case `default-major-mode` is `nil`. Modes such as Dired, Rmail, and Buffer List use this feature.

- 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 Sec-

tion 23.2.3 [Auto Major Mode], page 388). 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 207). If you do not autoload the mode command, it is sufficient to add the element in the file that contains the mode definition.

- In the comments that document the file, you should provide a sample `autoload` form and an example of how to add to `auto-mode-alist`, that users can include in their init files (see Section 39.1.2 [Init File], page 813).
- 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. Even if you never load the file more than once, someone else will.

### 23.2.3 How Emacs Chooses a Major Mode

Based on information in the file name or in the file itself, Emacs automatically selects a major mode for the new buffer when a file is visited. It also processes local variables specified in the file text.

#### `fundamental-mode`

[Command]

Fundamental mode is a major mode that is not specialized for anything in particular. Other major modes are defined in effect by comparison with this one—their definitions say what to change, starting from Fundamental mode. The `fundamental-mode` function does *not* run any mode hooks; you’re not supposed to customize it. (If you want Emacs to behave differently in Fundamental mode, change the *global* state of Emacs.)

#### `normal-mode &optional find-file`

[Command]

This function establishes the proper major mode and buffer-local variable bindings for the current buffer. First it calls `set-auto-mode` (see below), then it runs `hack-local-variables` to parse, and bind or evaluate as appropriate, the file’s local variables (see Section 11.13 [File Local Variables], page 155).

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.

If `normal-mode` processes the local variables list and this list specifies a major mode, that mode overrides any mode chosen by `set-auto-mode`. If neither `set-auto-mode` nor `hack-local-variables` specify a major mode, the buffer stays in the major mode determined by `default-major-mode` (see below).

`normal-mode` uses `condition-case` around the call to the major mode function, 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 the major mode that is appropriate for the current buffer. It bases its decision (in order of precedence) on the ‘`-*`’ line, 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*. However, this function does not look for the ‘`mode:`’ local variable near the end of a file; the `hack-local-variables` function does that. If `enable-local-variables` is `nil`, `set-auto-mode` does not check the ‘`-*`’ line for a mode tag either.

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.

**default-major-mode** [User Option]

This variable holds the default major mode for new buffers. The standard value is `fundamental-mode`.

If the value of `default-major-mode` is `nil`, Emacs uses the (previously) current buffer’s major mode as the default major mode of a new buffer. However, if that major mode symbol has a `mode-class` property with value `special`, then it is not used for new buffers; Fundamental mode is used instead. The modes that have this property are those such as `Dired` and `Rmail` that are useful only with text that has been specially prepared.

**set-buffer-major-mode** *buffer* [Function]

This function sets the major mode of *buffer* to the value of `default-major-mode`; if that variable 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 (`interpreter` . `mode`); for example, (“`perl`” . `perl-mode`) is one element present by default. The element says to use mode `mode` if the file specifies an interpreter which matches `interpreter`.

**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 25.8.4 [File Name Expansion], page 457), with version numbers and backup suffixes removed using `file-name-sans-versions` (see Section 25.8.1 [File Name Components], page 453), 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’.

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))
```

### 23.2.4 Getting Help about a Major Mode

The `describe-mode` function is used to provide information about major modes. It is normally called with `C-h m`. The `describe-mode` function uses the value of `major-mode`, which is why every major mode function needs to set the `major-mode` variable.

**describe-mode** [Command]

This function displays the documentation of the current major mode.

The `describe-mode` function calls the `documentation` function using the value of `major-mode` as an argument. Thus, it displays the documentation string of the major mode function. (See Section 24.2 [Accessing Documentation], page 426.)

**major-mode**

[Variable]

This buffer-local variable holds the symbol for the current buffer's major mode. This symbol should have a function definition that is the command to switch to that major mode. The `describe-mode` function uses the documentation string of the function as the documentation of the major mode.

### 23.2.5 Defining Derived Modes

It's often useful to define a new major mode in terms of an existing one. An easy way to do this is to use `define-derived-mode`.

**define-derived-mode** *variant parent name docstring keyword-args... body...* [Macro]

This construct 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. See Section 23.2.7 [Mode Hooks], page 393.

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.

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 docstring. If you omit *docstring*, `define-derived-mode` generates a documentation string.

The *keyword-args* are pairs of keywords and values. The values 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.)

**:group**

If this is specified, the value should be the customization group for this mode. (Not all major modes have one.) Only the (still experimental and unadvertised) command `customize-mode` currently uses this. `define-derived-mode` does *not* automatically define the specified customization group.

Here is a hypothetical example:

```
(define-derived-mode hypertext-mode
  text-mode "Hypertext"
  "Major mode for hypertext."
  \\{hypertext-mode-map}
  (setq case-fold-search nil))

(define-key hypertext-mode-map
  [down-mouse-3] 'do-hyper-link)
```

Do not write an `interactive` spec in the definition; `define-derived-mode` does that automatically.

### 23.2.6 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* [Macro]  
*auto-mode-list* *function-list* &**optional** *docstring*

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 35 [Syntax Tables], page 684.

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 23.6.2 [Search-based Fontification], page 414.

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`.

### 23.2.7 Mode Hooks

Every major mode function should finish by running its mode hook and the mode-independent 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 23.2.2 [Major Mode Conventions], page 385.

Emacs versions before Emacs 22 did not have `delay-mode-hooks`. When user-implemented major modes have not been updated to use it, 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 defined 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 23.1 [Hooks], page 382), but it also runs `after-change-major-mode-hook`.

When this function is called during the execution of a `delay-mode-hooks` form, it does not run the hooks 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.

#### `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 function.

### 23.2.8 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 text-mode-map
  (let ((map (make-sparse-keymap)))
    (define-key map "\e\t" 'ispell-complete-word)
    (define-key map "\es" 'center-line)
    (define-key map "\eS" 'center-paragraph)
    map)
  "Keymap for 'text-mode'.

Many other modes, such as Mail mode, Outline mode
and Indented Text mode, inherit all the commands
defined in this map.")
```

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'.
  (make-local-variable 'text-mode-variant)
  (setq text-mode-variant t)
  ;; These two lines are a feature added recently.
  (set (make-local-variable 'require-final-newline)
       mode-require-final-newline)
  (set (make-local-variable 'indent-line-function) 'indent-relative))
```

(The last line is redundant nowadays, since `indent-relative` is the default value, and we’ll delete it in a future version.)

Here is how it was defined formerly, before `define-derived-mode` existed:

```
;; This isn't needed nowadays, since define-derived-mode does it.
(defvar text-mode-abbrev-table nil
  "Abbrev table used while in text mode.")
(define-abbrev-table 'text-mode-abbrev-table ())

(defun text-mode ()
  "Major mode for editing text intended for humans to read...
  Special commands: \\{text-mode-map}
  Turning on text-mode runs the hook 'text-mode-hook'.
  (interactive)
  (kill-all-local-variables)
  (use-local-map text-mode-map)
  (setq local-abbrev-table text-mode-abbrev-table)
  (set-syntax-table text-mode-syntax-table))
```

```

;; These four lines are absent from the current version
;; not because this is done some other way, but rather
;; because nowadays Text mode uses the normal definition of paragraphs.
(make-local-variable 'paragraph-start)
(setq paragraph-start (concat "[ \t]*$\\|" page-delimiter))
(make-local-variable 'paragraph-separate)
(setq paragraph-separate paragraph-start)
(make-local-variable 'indent-line-function)
(setq indent-line-function 'indent-relative-maybe)
(setq mode-name "Text")
(setq major-mode 'text-mode)
(run-mode-hooks 'text-mode-hook)) ; Finally, this permits the user to
; customize the mode with a hook.

```

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.

```

;; Create mode-specific table variables.
(defvar lisp-mode-syntax-table nil "")
(defvar lisp-mode-abbrev-table nil "")

(defvar emacs-lisp-mode-syntax-table
  (let ((table (make-syntax-table)))
    (let ((i 0))

      ;; Set syntax of chars up to '0' to say they are
      ;; part of symbol names but not words.
      ;; (The digit '0' is 48 in the ASCII character set.)
      (while (< i ?0)
        (modify-syntax-entry i "_" table)
        (setq i (1+ i)))
      ;; ... similar code follows for other character ranges.
      ;; Then set the syntax codes for characters that are special in Lisp.
      (modify-syntax-entry ? " " table)
      (modify-syntax-entry ?\t " " table)
      (modify-syntax-entry ?\f " " table)
      (modify-syntax-entry ?\n ">" " table)
      ;; Give CR the same syntax as newline, for selective-display.
      (modify-syntax-entry ?\m ">" " table")
      (modify-syntax-entry ?\; "<" " table")
      (modify-syntax-entry ?, "," " table")
      (modify-syntax-entry ?, ";" " table")
      (modify-syntax-entry ?, "," " table")
      ;; ...likewise for many other characters...
      (modify-syntax-entry ?\(" () " table)
      (modify-syntax-entry ?\)" () " table)
      (modify-syntax-entry ?\[ " () " table)
      (modify-syntax-entry ?\]" () " table))
      table))
;; Create an abbrev table for lisp-mode.
(define-abbrev-table 'lisp-mode-abbrev-table ())

```

The three modes for Lisp share much of their code. For instance, each calls the following function to set various variables:

```

(defun lisp-mode-variables (lisp-syntax)
  (when lisp-syntax
    (set-syntax-table lisp-mode-syntax-table))
  (setq local-abbrev-table lisp-mode-abbrev-table)
  ...

```

In Lisp and most programming languages, we want the paragraph commands to treat only blank lines as paragraph separators. And the modes should understand the Lisp conventions for comments. The rest of `lisp-mode-variables` sets this up:

```
(make-local-variable 'paragraph-start)
(setq paragraph-start (concat page-delimiter "\\\\$"))
(make-local-variable 'paragraph-separate)
(setq paragraph-separate paragraph-start)
...
(make-local-variable 'comment-indent-function)
(setq comment-indent-function 'lisp-comment-indent))
...
```

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 shared-lisp-mode-map ()
  "Keymap for commands shared by all sorts of Lisp modes.")

;; Putting this if after the defvar is an older style.
(if shared-lisp-mode-map
  ()
  (setq shared-lisp-mode-map (make-sparse-keymap))
  (define-key shared-lisp-mode-map "\e\C-q" 'indent-sexp)
  (define-key shared-lisp-mode-map "\177"
    'backward-delete-char-untabify))
```

And here is the code to set up the keymap for Lisp mode:

```
(defvar lisp-mode-map ()
  "Keymap for ordinary Lisp mode...")

(if lisp-mode-map
  ()
  (setq lisp-mode-map (make-sparse-keymap))
  (set-keymap-parent lisp-mode-map shared-lisp-mode-map)
  (define-key lisp-mode-map "\e\C-x" 'lisp-eval-defun)
  (define-key lisp-mode-map "\C-c\C-z" 'run-lisp))
```

Finally, here is the complete major mode function definition for Lisp mode.

```
(defun lisp-mode ()
  "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.

Entry to this mode calls the value of 'lisp-mode-hook'
if that value is non-nil."
  (interactive)
  (kill-all-local-variables)
```

### 23.3 Minor Modes

A *minor mode* provides features that users may enable or disable independently of the choice of major mode. Minor modes can be enabled individually or in combination. Minor modes would be better named “generally available, optional feature modes,” except that such a name would be unwieldy.

A minor mode is not usually meant as a variation of a single major mode. Usually they are general and can apply to many major modes. For example, Auto Fill mode works with any major mode that permits text insertion. To be general, a minor mode must be effectively independent of the things major modes do.

A minor mode is often much more difficult to implement than a major mode. One reason is that you should be able to activate and deactivate minor modes in any order. A minor mode should be able to have its desired effect regardless of the major mode and regardless of the other minor modes in effect.

Often the biggest problem in implementing a minor mode is finding a way to insert the necessary hook into the rest of Emacs. Minor mode keymaps make this easier than it used to be.

`minor-mode-list` [Variable]

The value of this variable is a list of all minor mode commands.

### 23.3.1 Conventions for Writing Minor Modes

There are conventions for writing minor modes just as there are for major modes. Several of the major mode conventions apply to minor modes as well: those regarding the name of the mode initialization function, 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.

In addition, there are several conventions that are specific to minor modes. (The easiest way to follow all the conventions is to use the macro `define-minor-mode`; Section 23.3.3 [Defining Minor Modes], page 399.)

- Make a variable whose name ends in ‘`-mode`’ to control the minor mode. We call this the *mode variable*. The minor mode command should set this variable (`nil` to disable; anything else to enable).

If possible, implement the mode so that setting the variable automatically enables or disables the mode. Then the minor mode command does not need to do anything except set the variable.

This variable is used in conjunction with the `minor-mode-alist` to display the minor mode name in the mode line. It can also enable or disable a minor mode keymap. Individual commands or hooks can also check the variable's value.

If you want the minor mode to be enabled separately in each buffer, make the variable buffer-local.

- Define a command whose name is the same as the mode variable. Its job is to enable and disable the mode by setting the variable.

The command should accept one optional argument. If the argument is `nil`, it should toggle the mode (turn it on if it is off, and off if it is on). It should turn the mode on if the argument is a positive integer, the symbol `t`, or a list whose CAR is one of those. It should turn the mode off if the argument is a negative integer or zero, the symbol `-`, or a list whose CAR is a negative integer or zero. The meaning of other arguments is not specified.

Here is an example taken from the definition of `transient-mark-mode`. It shows the use of `transient-mark-mode` as a variable that enables or disables the mode's behavior, and also shows the proper way to toggle, enable or disable the minor mode based on the raw prefix argument value.

```
(setq transient-mark-mode
      (if (null arg) (not transient-mark-mode)
          (> (prefix-numeric-value arg) 0)))
```

- Add an element to `minor-mode-alist` for each minor mode (see [Definition of minor-mode-alist], page 406), 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)
  (setq minor-mode-alist
        (cons '(leif-mode " Leif") minor-mode-alist)))
```

or like this, using `add-to-list` (see Section 5.5 [List Variables], page 71):

```
(add-to-list 'minor-mode-alist '(leif-mode " Leif"))
```

Global minor modes distributed with Emacs should if possible support enabling and disabling via Custom (see Chapter 14 [Customization], page 185). To do this, the first step is to define the mode variable with `defcustom`, and specify `: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 207), and specify a `:require` so that customizing the variable will load the library that defines the mode. This will copy suitable definitions into ‘`loaddefs.el`’ so that users can use `customize-option` to enable 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)
```

### 23.3.2 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 356.

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. In general, this is the only way to do that, since 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.)

The key sequences bound in a minor mode should consist of `C-c` followed by one of `., /, ?, " , ] , \ , ^ , ! , # , $ , % , & , * , ( , ) , - , + , =`. (The other punctuation characters are reserved for major modes.)

### 23.3.3 Defining Minor Modes

The macro `define-minor-mode` offers a convenient way of implementing a mode in one self-contained definition.

<code>define-minor-mode mode doc [init-value [lighter [keymap]]]</code>	[Macro]
<code>keyword-args... body...</code>	

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. It also defines a variable named *mode*, which is set to `t` or `nil` by enabling or disabling the mode. The variable is initialized to *init-value*. Except in unusual circumstances (see below), this value must be `nil`.

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.

The optional argument *keymap* specifies the keymap for the minor mode. It can be a variable name, whose value is the keymap, or it can be an alist specifying bindings in this form:

```
(key-sequence . definition)
```

The above three arguments *init-value*, *lighter*, and *keymap* can be (partially) omitted when *keyword-args* are used. The *keyword-args* consist of keywords followed by corresponding values. A few keywords have special meanings:

**:group group**

Custom group name to use in all generated `defcustom` forms. Defaults to *mode* without the possible trailing ‘`-mode`’. **Warning:** don’t use this default group name unless you have written a `defgroup` to define that group properly. See Section 14.2 [Group Definitions], page 187.

**: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 Custom 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 `define-minor-mode` form is evaluated each time Emacs starts; for packages that are not part of Emacs, the easiest way to do this is to specify a `:require` keyword.

**:init-value init-value**

This is equivalent to specifying *init-value* positionally.

**:lighter lighter**

This is equivalent to specifying *lighter* positionally.

**:keymap keymap**

This is equivalent to specifying *keymap* positionally.

Any other keyword arguments are passed directly to the `defcustom` generated for the variable *mode*.

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 finishes by running the mode hook variable `mode-hook`.

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.
With no argument, this command toggles the mode.
Non-null prefix argument turns on the mode.
Null prefix argument turns off the mode."
```

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-\^?" . hungry-electric-delete))
:group 'hunger)

```

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`. It puts the variable `hungry-mode` into custom group `hunger`. 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.
  With no argument, this command toggles the mode.
  Non-null prefix argument turns on the mode.
  Null prefix argument turns off the mode.

  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.
  :init-value nil
  ;; The indicator for the mode line.
  :lighter " Hungry"
  ;; The minor mode bindings.
  :keymap
  '(("C-\^?" . hungry-electric-delete)
    ("C-\M-\^?"
     . (lambda ()
        (interactive)
        (hungry-electric-delete t))))
  :group 'hunger)

```

`define-globalized-minor-mode global-mode mode turn-on  
keyword-args...`

[Macro]

This defines a global toggle named `global-mode` whose meaning is to enable or disable the buffer-local minor mode `mode` in all buffers. 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.

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 defines the customization option `global-mode` (see Chapter 14 [Customization], page 185), which can be toggled in the Custom 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.

## 23.4 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.

### 23.4.1 Mode Line Basics

`mode-line-format` is a buffer-local variable that holds a *mode line construct*, a kind of template, which controls what is displayed on the mode line of the current buffer. 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`.

For efficiency, Emacs does not continuously recompute the mode line and header line of a window. 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 change the buffer’s modification status. If you modify any of the variables referenced by `mode-line-format` (see Section 23.4.4 [Mode Line Variables], page 405), or any other variables and data structures that affect how text is displayed (see Chapter 38 [Display], page 739), you may want to force an update of the mode line so as to display the new information or display it in the new way.

**force-mode-line-update** &optional all [Function]

Force redisplay of the current buffer’s mode line and header line. The next redisplay will update the mode line and header line based on the latest values of all relevant variables. With optional non-`nil` `all`, force redisplay of all mode lines and header lines.

This function also forces recomputation of the menu bar menus and the frame title.

The selected window’s mode line is usually displayed in a different color using the face `mode-line`. Other windows’ mode lines appear in the face `mode-line-inactive` instead. See Section 38.12 [Faces], page 762.

### 23.4.2 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 29.4 [Frame Titles], page 540) and header lines (see Section 23.4.7 [Header Lines], page 409).

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 23.4.5 [%-Constructs], page 407.

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 **local-map** properties in *string* have special meanings. See Section 23.4.6 [Properties in Mode], page 409.

**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.

**(: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*. (This feature is new as of Emacs 22.1.)

**(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 element. Otherwise, the third element, *else*, is processed recursively. You may omit *else*; then the mode-line element 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").

### 23.4.3 The Top Level of Mode Line Control

The variable in overall control of the mode line is `mode-line-format`.

`mode-line-format`

[Variable]

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 never displays 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 23.4.4 [Mode Line Variables], page 405), 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 an example of a `mode-line-format` that might be useful for `shell-mode`, since it contains the host name and default directory.

```
(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 (mode-line-mode-name)))
```

```
'mode-line-process
'minor-mode-alist
"%n"
")%]--"
'(which-func-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-func-mode` enable particular minor modes; as usual, these variable names are also the minor mode command names.)

#### 23.4.4 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 `mode-line-format`'s value were 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.

##### `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 33 [Non-ASCII Characters], page 640.

##### `mode-line-modified`

[Variable]

This variable holds the value of the mode-line construct that displays whether the current buffer is modified.

The default value of `mode-line-modified` is ("%"1\*%"1+"). This means that the mode line 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. The default value is " " 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 is ("%12b"), which displays the buffer name, padded with spaces to at least 12 columns.

##### `mode-line-position`

[Variable]

This variable indicates the position in the buffer. Here is a simplified version of its default value. The actual default value also specifies addition of the `help-echo` text property.

```
((-3 "%p")
 (size-indication-mode (8 " of %I"))
 (line-number-mode
  ((column-number-mode
   (10 " (%l,%c)")
   (6 " L%l")))
  ((column-number-mode
   (5 " C%c")))))
```

This means that `mode-line-position` displays at least the buffer percentage and possibly the buffer size, the line number and the column number.

**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. Here is a simplified version of its default value. The real default value also specifies addition of text properties.

```
("%" [" mode-name
 mode-line-process minor-mode-alist
 "%n" ")]--")
```

So `mode-line-modes` normally also displays the recursive editing level, information on the process status and whether narrowing is in effect.

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.

**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`.

**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:

$$(\text{minor-mode-variable} \text{ mode-line-string})$$

More generally, `mode-line-string` can be any mode-line spec. 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 spec that, by default, appears in the mode line just after the *which-func-mode* minor mode if set, else after *mode-line-modes*. 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*, but that is obsolete, since the variable is included in the mode line from *mode-line-format*.

The variable *default-mode-line-format* is where *mode-line-format* usually gets its value:

#### **default-mode-line-format** [Variable]

This variable holds the default *mode-line-format* for buffers that do not override it. This is the same as (*default-value* 'mode-line-format).

Here is a simplified version of the default value of *default-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-func-mode (" " which-func-format "--"))
 (global-mode-string ("--" global-mode-string))
 "-%-")
```

#### **23.4.5 %-Constructs in the Mode Line**

Strings used as mode-line constructs can use certain %-constructs to substitute various kinds of data. Here 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 with spaces to the right.

%b	The current buffer name, obtained with the <i>buffer-name</i> function. See Section 27.3 [Buffer Names], page 484.
%c	The current column number of point.
%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 <code>buffer-file-name</code> function. See Section 27.4 [Buffer File Name], page 485.
%F	The title (only on a window system) or the name of the selected frame. See Section 29.3.3.1 [Basic Parameters], page 533.
%i	The size of the accessible part of the current buffer; basically <code>(- (point-max) (point-min))</code> .
%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.
%n	'Narrow' when narrowing is in effect; nothing otherwise (see <code>narrow-to-region</code> in Section 30.4 [Narrowing], page 569).
%p	The percentage of the buffer text above the <code>top</code> of window, or 'Top', 'Bottom' or 'All'. Note that the default mode-line specification truncates this to three characters.
%P	The percentage of the buffer text that is above the <code>bottom</code> 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'.
%s	The status of the subprocess belonging to the current buffer, obtained with <code>process-status</code> . See Section 37.6 [Process Information], page 712.
%t	Whether the visited file is a text file or a binary file. This is a meaningful distinction only on certain operating systems (see Section 33.10.9 [MS-DOS File Types], page 658).
%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 <code>buffer-read-only</code> ); '*' if the buffer is modified (see <code>buffer-modified-p</code> ); '-' otherwise. See Section 27.5 [Buffer Modification], page 487.
%+	'*' if the buffer is modified (see <code>buffer-modified-p</code> ); '%' if the buffer is read only (see <code>buffer-read-only</code> ); '-' otherwise. This differs from '%*' only for a modified read-only buffer. See Section 27.5 [Buffer Modification], page 487.
%&	'*' if the buffer is modified, and '-' otherwise.
%[	An indication of the depth of recursive editing levels (not counting minibuffer levels): one '[' for each editing level. See Section 21.12 [Recursive Editing], page 342.
%]	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.

The following two %-constructs are still supported, but they are obsolete, since you can get the same results with the variables `mode-name` and `global-mode-string`.

- %m        The value of `mode-name`.
- %M        The value of `global-mode-string`.

### 23.4.6 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 `local-map` 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.
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 `local-map` 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.

### 23.4.7 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 different variables.

**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 23.4.2 [Mode Line Data], page 402).

**default-header-line-format** [Variable]

This variable holds the default `header-line-format` for buffers that do not override it. This is the same as `(default-value 'header-line-format)`.

It is normally `nil`, so that ordinary buffers have no header line.

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.

### 23.4.8 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 specification.

`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 instead of displaying the text in the mode line or the header line, it 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. And any character for which no `face` property is specified gets a default value which is usually `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 no `face` property.)

However, if `face` is an integer, the value has no text properties.

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.

## 23.5 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` [Function]

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 34.3 [Regular Expressions], page 663); 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\\)\\\n  \s+([-A-Za-z0-9]+)\") 2)\n  ("*Vars*" "^\s-*(def\\(var\\|const\\)\\\n  \s+([-A-Za-z0-9]+)\") 2)\n  ("*Types*"\n    "\^\\s-\"\n    (def\\(type\\|struct\\|class\\|ine-condition\\)\\\n      \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 35.3 [Syntax Table Functions], page 688).

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.

## 23.6 Font Lock Mode

*Font Lock mode* is a feature 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 two ways: through syntactic parsing based on the syntax table, and through searching (usually for regular expressions). Syntactic

fontification happens first; it finds comments and string constants and highlights them. Search-based fontification happens second.

### 23.6.1 Font Lock Basics

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 major modes, as a buffer-local variable, to specify how to fontify text in that mode. It automatically becomes buffer-local when you set it. 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 [syntax-begin 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 how to do level 1 fontification, the second symbol how to do level 2, and so on. See Section 23.6.5 [Levels of Font Lock], page 419.

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`, such fontification is not performed. See Section 23.6.8 [Syntactic Font Lock], page 420.

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 when searching as directed by `font-lock-keywords`.

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 (see Section 35.3 [Syntax Table Functions], page 688). The resulting syntax table is stored in `font-lock-syntax-table`.

The fifth element, `syntax-begin`, specifies the value of `font-lock-beginning-of-syntax-function`. We recommend setting this variable to `nil` and using `syntax-begin-function` instead.

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 23.6.4 [Other Font Lock Variables], page 418.

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.

### 23.6.2 Search-based Fontification

The most important variable for customizing Font Lock mode is `font-lock-keywords`. It specifies the search criteria for search-based fontification. You should specify the value of this variable with `keywords` in `font-lock-defaults`.

**`font-lock-keywords`** [Variable]

This variable's value is a list of the keywords to highlight. Be careful when composing regular expressions for this list; a poorly written pattern can dramatically slow things down!

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\\>"
```

The function `regexp-opt` (see Section 34.3.3 [Regexp Functions], page 672) is useful for calculating optimal regular expressions to match a number of different 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)
```

If you use `regexp-opt` to produce the regular expression `matcher`, you can use `regexp-opt-depth` (see Section 34.3.3 [Regexp Functions], page 672) to calculate the value for `subexp`.

**(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:

```
(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 23.6.4 [Other Font Lock Variables], page 418.

**(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 See Section 23.6.10 [Multiline Font Lock], page 422.

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.

### 23.6.3 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.

`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 functions 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.

For example, this code

```
(font-lock-add-keywords 'c-mode
  '(((\\<\\(FIXME\\):" 1 font-lock-warning-face prepend)
    ("\\<\\(and\\|or\\|not\\)\\>" . font-lock-keyword-face)))
```

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.

That 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)))))
```

### 23.6.4 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 23.6.1 [Font Lock Basics], page 413).

#### `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-o M-o (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 23.6.2 [Search-based Fontification], page 414.

#### `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`.

### 23.6.5 Levels of Font Lock

Many 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. 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.

### 23.6.6 Precalculated Fontification

In addition to using `font-lock-defaults` for search-based fontification, you may use the special character property `font-lock-face` (see Section 32.19.4 [Special Properties], page 620). This property acts just like the explicit `face` property, but its activation is toggled when the user calls `M-x font-lock-mode`. Using `font-lock-face` is especially convenient for special modes which construct their text programmatically, such as `list-buffers` and `occur`.

If your mode does not use any of the other machinery of Font Lock (i.e. it only uses the `font-lock-face` property), it should not set the variable `font-lock-defaults`.

### 23.6.7 Faces for Font Lock

You can make Font Lock mode use any face, but several faces are defined specifically for Font Lock mode. 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`. This means you can write `font-lock-comment-face` in a context such as `font-lock-keywords` where a face-name-valued expression is used.

#### `font-lock-comment-face`

Used (typically) for comments.

#### `font-lock-comment-delimiter-face`

Used (typically) for comments delimiters.

#### `font-lock-doc-face`

Used (typically) for documentation strings in the code.

#### `font-lock-string-face`

Used (typically) for string constants.

#### `font-lock-keyword-face`

Used (typically) for keywords—names that have special syntactic significance, like `for` and `if` in C.

**font-lock-builtin-face**

Used (typically) for built-in function names.

**font-lock-function-name-face**

Used (typically) for the name of a function being defined or declared, in a function definition or declaration.

**font-lock-variable-name-face**

Used (typically) for the name of a variable being defined or declared, in a variable definition or declaration.

**font-lock-type-face**

Used (typically) for names of user-defined data types, where they are defined and where they are used.

**font-lock-constant-face**

Used (typically) for constant names.

**font-lock-preprocessor-face**

Used (typically) for preprocessor commands.

**font-lock-negation-char-face**

Used (typically) for easily-overlooked negation characters.

**font-lock-warning-face**

Used (typically) for constructs that are peculiar, or that greatly change the meaning of other text. For example, this is used for ‘;; ;###autoload’ cookies in Emacs Lisp, and for #error directives in C.

### 23.6.8 Syntactic Font Lock

Syntactic fontification uses the syntax table to find comments and string constants (see Chapter 35 [Syntax Tables], page 684). It highlights them using `font-lock-comment-face` and `font-lock-string-face` (see Section 23.6.7 [Faces for Font Lock], page 419), or whatever `font-lock-syntactic-face-function` chooses. There are several variables that affect syntactic fontification; you should set them by means of `font-lock-defaults` (see Section 23.6.1 [Font Lock Basics], page 413).

**font-lock-keywords-only**

[Variable]

`Non-nil` means Font Lock should not do syntactic fontification; it should only fontify based on `font-lock-keywords`. The normal way for a mode to set this variable to `t` is with `keywords-only` in `font-lock-defaults`.

**font-lock-syntax-table**

[Variable]

This variable holds the syntax table to use for fontification of comments and strings. Specify it using `syntax-alist` in `font-lock-defaults`. If this is `nil`, fontification uses the buffer’s syntax table.

**font-lock-beginning-of-syntax-function**

[Variable]

If this variable is `non-nil`, it should be a function to move point back to a position that is syntactically at “top level” and outside of strings or comments. Font Lock uses this when necessary to get the right results for syntactic fontification.

This function is called with no arguments. It should leave point at the beginning of any enclosing syntactic block. Typical values are `beginning-of-line` (used when the start of the line is known to be outside a syntactic block), or `beginning-of-defun` for programming modes, or `backward-paragraph` for textual modes.

If the value is `nil`, Font Lock uses `syntax-begin-function` to move back outside of any comment, string, or sexp. This variable is semi-obsolete; we recommend setting `syntax-begin-function` instead.

Specify this variable using `syntax-begin` in `font-lock-defaults`.

#### `font-lock-syntactic-face-function`

[Variable]

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.

This can be used to highlighting different kinds of strings or comments differently. It is also sometimes abused together with `font-lock-syntactic-keywords` to highlight constructs that span multiple lines, but this is too esoteric to document here.

Specify this variable using `other-vars` in `font-lock-defaults`.

### 23.6.9 Setting Syntax Properties

Font Lock mode can be used to update `syntax-table` properties automatically (see Section 35.4 [Syntax Properties], page 690). This is useful in languages for which a single syntax table by itself is not sufficient.

#### `font-lock-syntactic-keywords`

[Variable]

This variable enables and controls updating `syntax-table` properties by Font Lock. Its value should be a list of elements of this form:

`(matcher subexp syntax override laxmatch)`

The parts of this element have the same meanings as in the corresponding sort of element of `font-lock-keywords`,

`(matcher subexp facespec override laxmatch)`

However, instead of specifying the value `facespec` to use for the `face` property, it specifies the value `syntax` to use for the `syntax-table` property. Here, `syntax` can be a string (as taken by `modify-syntax-entry`), a syntax table, a cons cell (as returned by `string-to-syntax`), or an expression whose value is one of those two types. `override` cannot be `prepend` or `append`.

For example, an element of the form:

`("\\$\\\\(#\\\" 1 ".")`

highlights syntactically a hash character when following a dollar character, with a SYNTAX of `". "` (meaning punctuation syntax). Assuming that the buffer syntax table specifies hash characters to have comment start syntax, the element will only highlight hash characters that do not follow dollar characters as comments syntactically.

An element of the form:

```
("\\"(\\"\").\\\"(\\"\")"
(1 \"\"")
(2 \"\""))
```

highlights syntactically both single quotes which surround a single character, with a SYNTAX of "\\" (meaning string quote syntax). Assuming that the buffer syntax table does not specify single quotes to have quote syntax, the element will only highlight single quotes of the form ‘‘c’’ as strings syntactically. Other forms, such as ‘‘foo’bar’ or ‘‘fubar’’, will not be highlighted as strings.

Major modes normally set this variable with `other-vars` in `font-lock-defaults`.

### 23.6.10 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 three 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.

### 23.6.10.1 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 the requires that Font-Lock 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 looks at it, or use `font-lock-fontify-region-function`.

### 23.6.10.2 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 fontify by setting one the following variables:

#### `font-lock-extend-after-change-region-function`

[Variable]

This buffer-local variable is either `nil` or a function for Font-Lock 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 32.26 [Change Hooks], page 638). 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.

## 23.7 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`.

## 24 Documentation

GNU Emacs Lisp has convenient on-line help facilities, most of which derive their information from the documentation strings associated with functions and variables. This chapter describes how to write good documentation strings for your Lisp programs, as well as how to write programs to access documentation.

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*. For the conventions for writing documentation strings, see Section D.6 [Documentation Tips], page 862.

### 24.1 Documentation Basics

A documentation string is written using the Lisp syntax for strings, with double-quote characters surrounding the text of the string. This is because it really is a Lisp string object. The string serves as documentation when it is written in the proper place in the definition of a function or variable. In a function definition, the documentation string follows the argument list. In a variable definition, the documentation string follows the initial value of the variable.

When you write a documentation string, make the first line a complete sentence (or two complete sentences) since some commands, such as `apropos`, show only the first line of a multi-line documentation string. Also, you should not indent the second line of a documentation string, if it has one, because that looks odd when you use `C-h f (describe-function)` or `C-h v (describe-variable)` to view the documentation string. There are many other conventions for doc strings; see Section D.6 [Documentation Tips], page 862.

Documentation strings can contain several special substrings, which stand for key bindings to be looked up in the current keymaps when the documentation is displayed. This allows documentation strings to refer to the keys for related commands and be accurate even when a user rearranges the key bindings. (See Section 24.3 [Keys in Documentation], page 428.)

Emacs Lisp mode fills documentation strings to the width specified by `emacs-lisp-docstring-fill-column`.

In Emacs Lisp, a documentation string is accessible through the function or variable that it describes:

- The documentation for a function is usually stored in the function definition itself (see Section 12.2 [Lambda Expressions], page 161). The function `documentation` knows how to extract it. You can also put function documentation in the `function-documentation` property of the function name. That is useful with definitions such as keyboard macros that can't hold a documentation string.
- The documentation for a variable is stored in the variable's property list under the property name `variable-documentation`. The function `documentation-property` knows how to retrieve it.

To save space, the documentation for preloaded functions and variables (including primitive functions and autoloaded functions) is stored in the file ‘`emacs/etc/DOC-version`’—not inside Emacs. The documentation strings for functions and variables loaded during the Emacs session from byte-compiled files are stored in those files (see Section 16.3 [Docs and Compilation], page 217).

The data structure inside Emacs has an integer offset into the file, or a list containing a file name and an integer, in place of the documentation string. The functions `documentation` and `documentation-property` use that information to fetch the documentation string from the appropriate file; this is transparent to the user.

The ‘`emacs/lib-src`’ directory contains two utilities that you can use to print nice-looking hardcopy for the file ‘`emacs/etc/DOC-version`’. These are ‘`sorted-doc`’ and ‘`digest-doc`’.

## 24.2 Access to Documentation Strings

`documentation-property symbol property &optional verbatim` [Function]

This function returns the documentation string that is recorded in `symbol`’s property list under property `property`. It retrieves the text from a file if the value calls for that. If the property value isn’t `nil`, isn’t a string, and doesn’t refer to text in a file, then it is evaluated to obtain a string.

The last thing this function does is pass the string through `substitute-command-keys` to substitute actual key bindings, unless `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`. `documentation` 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`, it calls `substitute-command-keys` so as to return a value containing the actual (current) key bindings.

The function `documentation` 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`.

`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
         (function
          (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 (user-variable-p s)
                             "Option " "Variable")
                         (or (documentation-property
                               s 'variable-documentation)
                             "not documented"))))))
              sym-list)

  ;; Build a list of symbols that match pattern.
  (mapatoms (function
              (lambda (sym)
                (if (string-match pattern (symbol-name sym))
                    (setq sym-list (cons sym sym-list))))))

  ;; Display the data.
  (with-output-to-temp-buffer "*Help*"
    (mapcar describe-func (sort sym-list 'string<))
    (print-help-return-message))))
```

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 ...

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, 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'.
```

```
temporary-goal-column  Variable
Current goal column for vertical motion.
It is the column where point was
at the start of current run of vertical motion commands.
When the 'track-eol' feature is doing its job, the value is 9999.
----- Buffer: *Help* -----
```

The asterisk '\*' as the first character of a variable's doc string, as shown above for the `goal-column` variable, means that it is a user option; see the description of `defvar` in Section 11.5 [Defining Variables], page 139.

#### **Snarf-documentation** *filename*

[Function]

This function is used only during Emacs initialization, just before the runnable Emacs is dumped. It finds the file offsets of the documentation strings stored in the file *filename*, and records them in the in-core function definitions and variable property lists in place of the actual strings. See Section E.1 [Building Emacs], page 870.

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-version`".

#### **doc-directory**

[Variable]

This variable holds the name of the directory which should contain the file "`DOC-version`" that contains documentation strings for built-in and preloaded 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 433.

In older Emacs versions, `exec-directory` was used for this.

### 24.3 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`.

\<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.

\=

quotes the following character and is discarded; thus, ‘\=\[’ puts ‘\[’ into the output, and ‘\=\=’ puts ‘\=’ into the output.

**Please note:** Each ‘\’ must be doubled when written in a string in Emacs Lisp.

**substitute-command-keys** *string*

[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.

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
"

(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."
```

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 862, for details.

## 24.4 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 21.6 [Input Events], page 315, 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-1 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 appear in text—like `single-key-description`, except that control characters are represented with a leading caret (which is how control characters in Emacs buffers are usually displayed). Another difference is that `text-char-description` recognizes the 2\*\*7 bit as the Meta character, whereas `single-key-description` uses the 2\*\*27 bit for Meta.

```
(text-char-description ?\C-c)
  ⇒ "^C"
(text-char-description ?\M-m)
  ⇒ "\xed"
(text-char-description ?\C-\M-m)
  ⇒ "\x8d"
(text-char-description (+ 128 ?m))
  ⇒ "M-m"
(text-char-description (+ 128 ?\C-m))
  ⇒ "M-^M"
```

**read-kbd-macro** *string &optional need-vector* [Function]

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 22.1 [Key Sequences], page 347.) If *need-vector* is non-*nil*, the return value is always a vector.

## 24.5 Help Functions

Emacs provides a variety of on-line 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 fn-doc var-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, and the remaining elements are documentation strings for *symbol*’s various roles (or *nil*).

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:

```
(define-key global-map (char-to-string help-char) 'help-command)
(fset 'help-command help-map)
```

**print-help-return-message &optional function** [Function]

This function builds a string that explains how to restore the previous state of the windows after a help command. After building the message, it applies *function* to it if *function* is non-*nil*. Otherwise it calls *message* to display it in the echo area.

This function expects to be called inside a *with-output-to-temp-buffer* special form, and expects *standard-output* to have the value bound by that special form. For an example of its use, see the long example in Section 24.2 [Accessing Documentation], page 426.

**help-char** [Variable]

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` [Variable]

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` or `read-char` 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 302).

#### `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` [Function]

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. In older Emacs versions, `exec-directory` was used for this.

**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 Files

In Emacs, you can find, create, view, save, and otherwise work with files and file directories. This chapter describes most of the file-related functions of Emacs Lisp, but a few others are described in Chapter 27 [Buffers], page 481, and those related to backups and auto-saving are described in Chapter 26 [Backups and Auto-Saving], page 471.

Many of the file functions take one or more arguments that are file names. A file name is actually a string. Most of these functions expand file name arguments by calling `expand-file-name`, so that ‘~’ is handled correctly, as are relative file names (including ‘..’). These functions don’t recognize environment variable substitutions such as ‘\$HOME’. See Section 25.8.4 [File Name Expansion], page 457.

When file I/O functions signal Lisp errors, they usually use the condition `file-error` (see Section 10.5.3.3 [Handling Errors], page 129). The error message is in most cases obtained from the operating system, according to locale `system-message-locale`, and decoded using coding system `locale-coding-system` (see Section 33.12 [Locales], page 660).

### 25.1 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). Usually, a buffer contains information that you have copied from a file; then we say the buffer is visiting that file. The copy in the buffer is what you modify with editing commands. Such changes to the buffer do not change the file; therefore, to make the changes permanent, you must save the buffer, which means copying the altered buffer contents back into the file.

In spite of 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.

#### 25.1.1 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 27.4 [Buffer File Name], page 485, 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 25.3 [Reading from Files], page 440.

`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 28.7 [Displaying Buffers], page 506.)

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-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 33.10 [Coding Systems], page 648), including end-of-line conversion, and format conversion (see Section 25.12 [Format Conversion], page 468). 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 25.1.2 [Subroutines of Visiting], page 436). 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 28.7 [Displaying Buffers], page 506.

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 27.7 [Read Only Buffers], page 489, for related functions and variables.

When this command is called interactively, it prompts for *filename*.

**view-file** *filename* [Command]

This command visits *filename* using View mode, returning to the previous buffer when you exit View mode. View mode is a minor mode that provides commands to skim rapidly through the file, but does not let you modify the text. Entering View mode runs the normal hook **view-mode-hook**. See Section 23.1 [Hooks], page 382.

When **view-file** 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** [Variable]

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 23.1 [Hooks], page 382.

**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.

### 25.1.2 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 27.9 [Creating Buffers], page 492.

**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 27.9 [Creating Buffers], page 492).

**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 23.2.3 [Auto Major Mode], page 388). It is called by `find-file-noselect` and by the default revert function (see Section 26.3 [Reverting], page 479).

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`.

## 25.2 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.

**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 controls which buffers to ask about (or to save silently if *save-silently-p* is non-*nil*). If it is *nil*, that means to 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 27.10 [Killing Buffers], page 493). 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 *pred* 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 27.4 [Buffer File Name], page 485) 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 an existing directory, or a symbolic link to one, `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 25.12 [Format Conversion], page 468), and may save text properties in “annotations” (see Section 32.19.7 [Saving Properties], page 626).

**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 26.1.1 [Making Backup Files], page 471.

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 33.10.3 [Lisp and Coding Systems], page 650), perform the encoding (see Section 33.10.7 [Explicit Encoding], page 656), and set `last-coding-system-used` to the coding system that was used (see Section 33.10.2 [Encoding and I/O], page 649).

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 23.1 [Hooks], page 382.

**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. 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`.

**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. One use of this hook is in Fast Lock mode; it uses this hook to save the highlighting information in a cache 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 26.1.2 [Rename or Copy], page 473. 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 file whenever the buffer being saved does not already end in one. If the value of the variable is `non-nil`, but not `t`, then `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 27.4 [Buffer File Name], page 485).

### 25.3 Reading from Files

You can copy a file from the disk and insert it into a buffer using the `insert-file-contents` function. Don't use the user-level 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.

The function `insert-file-contents` checks the file contents against the defined file formats, and converts the file contents if appropriate. See Section 25.12 [Format Conversion], page 468. It also calls the functions in the list `after-insert-file-functions`; see Section 32.19.7 [Saving Properties], page 626. Normally, one of the functions in the `after-insert-file-functions` list determines the coding system (see Section 33.10 [Coding Systems], page 648) used for decoding the file's contents, including end-of-line conversion.

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 integers 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 first 500 characters of a file.

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` are `nil`.

**insert-file-contents-literally** *filename* &optional *visit beg end* [Function]  
*replace*

This function works like **insert-file-contents** except that it does not do format decoding (see Section 25.12 [Format Conversion], page 468), does not do character code conversion (see Section 33.10 [Coding Systems], page 648), does not run **find-file-hook**, does not perform 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 25.11 [Magic File Names], page 464.

## 25.4 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 *filename* specifies a nonwritable file, or a nonexistent file in a directory where files cannot be created.

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 an integer, **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.

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**. See Section 25.12 [Format Conversion], page 468. It also calls the functions in the list **write-region-annotate-functions**; see Section 32.19.7 [Saving Properties], page 626.

Normally, **write-region** displays the message ‘Wrote *filename*’ in the echo area. If *visit* is neither **t** nor **nil** nor a string, then this message is inhibited. This feature is useful for programs that use files for internal purposes, files that the user does not need to know about.

#### **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 10.5 [Nonlocal Exits], page 125).

See also **with-temp-buffer** in [The Current Buffer], page 483.

## 25.5 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. (File locks are not implemented on Microsoft systems.) 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.

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 27.6 [Modification Time], page 488.

#### **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 system does not support locking.

**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 if the system does not support locking.

File locking is not supported on some systems. On systems that do not support it, the functions `lock-buffer`, `unlock-buffer` and `file-locked-p` do nothing and return `nil`.

**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. The code for its usual definition is in '`userlock.el`'.

## 25.6 Information about Files

The functions described in this section all operate on strings that designate file names. With a few exceptions, all the functions have names that begin with the word 'file'. These functions all return information about actual files or directories, so their arguments must all exist as actual files or directories unless otherwise noted.

### 25.6.1 Testing Accessibility

These functions test for permission to access a file in specific ways. Unless explicitly stated otherwise, they recursively follow symbolic links for their file name arguments, at all levels (at the level of the file itself and at all levels of parent directories).

**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 find out its attributes. (On Unix

and GNU/Linux, this is true if the file exists and you have execute permission on the containing directories, regardless of the protection of the file itself.)

If the file does not exist, or if fascist access control policies prevent you from finding the attributes of the file, this function returns `nil`.

Directories are files, so `file-exists-p` returns `t` when given a directory name. However, symbolic links are treated specially; `file-exists-p` returns `t` for a symbolic link name only if the target file exists.

**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-readable-p "files.texi")
  ⇒ t
(file-exists-p "/usr/spool/mqueue")
  ⇒ t
(file-readable-p "/usr/spool/mqueue")
  ⇒ nil
```

**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 Unix and GNU/Linux, 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 the specified directory does exist and you can write in that directory.

In the third 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 "~/foo")
  ⇒ t
(file-writable-p "/foo")
  ⇒ nil
(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 (or 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).

Example: after the following,

```
(file-accessible-directory-p "/foo")
  ⇒ nil
```

we can deduce that any attempt to read a file in ‘`/foo/`’ will give an error.

**access-file** *filename string* [Function]

This function opens file *filename* for reading, then closes it and returns **nil**. However, if the open fails, it signals an error using *string* as the error message text.

**file-ownership-preserved-p** *filename* [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 *filename* is a symbolic link, then, unlike the other functions discussed here, **file-ownership-preserved-p** does *not* replace *filename* with its target. However, it does recursively follow symbolic links at all levels of parent directories.

**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
```

You can use **file-attributes** to get a file's last modification time as a list of two numbers. See Section 25.6.4 [File Attributes], page 447.

### 25.6.2 Distinguishing Kinds of Files

This section describes how to distinguish various kinds of files, such as directories, symbolic links, and ordinary files.

**file-symlink-p** *filename* [Function]

If the file *filename* is a symbolic link, the **file-symlink-p** function returns the (non-recursive) link target as a string. (Determining the file name that the link points to from the target is nontrivial.) First, this function recursively follows symbolic links at all levels of parent directories.

If the file *filename* is not a symbolic link (or there is no such file), **file-symlink-p** returns **nil**.

```
(file-symlink-p "foo")
  ⇒ nil
(file-symlink-p "sym-link")
  ⇒ "foo"
(file-symlink-p "sym-link2")
  ⇒ "sym-link"
(file-symlink-p "/bin")
  ⇒ "/pub/bin"
```

The next two functions recursively follow symbolic links at all levels for *filename*.

**file-directory-p** *filename* [Function]

This function returns **t** if *filename* is the name of an existing directory, **nil** otherwise.

```
(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).

### 25.6.3 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]

The function **file-truename** returns the truename of the file *filename*. The argument must be an absolute file name.

This function does not expand environment variables. Only **substitute-in-file-name** does that. See [Definition of substitute-in-file-name], page 458.

If you may need to follow symbolic links preceding ‘..’ appearing as a name component, you should make sure to call **file-truename** without prior direct or indirect calls to **expand-file-name**, as 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. See Section 25.8.4 [Functions that Expand Filenames], page 457.

**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"
```

See Section 27.4 [Buffer File Name], page 485, for related information.

#### 25.6.4 Other Information about Files

This section describes the functions for getting detailed information about a file, other than its contents. This information includes the mode bits that control access permission, the owner and group numbers, the number of names, the inode number, the size, and the times of access and modification.

**file-modes** *filename* [Function]

This function returns the mode bits of *filename*, as an integer. The mode bits are also called the file permissions, and they specify access control in the usual Unix fashion. If the low-order bit is 1, then the file is executable by all users, if the second-lowest-order bit is 1, then the file is writable by all users, etc.

The highest value returnable is 4095 (7777 octal), meaning that everyone has read, write, and execute permission, that the SUID bit is set for both others and group, and that the sticky bit is set.

If *filename* does not exist, `file-modes` returns `nil`.

This function recursively follows symbolic links at all levels.

```
(file-modes "~/junk/diffs")
⇒ 492 ; Decimal integer.
(format "%o" 492)
⇒ "754" ; Convert to octal.

(set-file-modes "~/junk/diffs" 438)
⇒ nil

(format "%o" 438)
⇒ "666" ; Convert to octal.

% ls -l diffs
-rw-rw-rw- 1 lewis 0 3063 Oct 30 16:00 diffs
```

If the *filename* argument to the next two functions is a symbolic link, then these function do *not* replace it with its target. However, they both recursively follow symbolic links at all levels of parent directories.

**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, then 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.

```
% ls -l foo*
-rw-rw-rw- 2 rms      4 Aug 19 01:27 foo
-rw-rw-rw- 2 rms      4 Aug 19 01:27 foo1

(file-nlinks "foo")
⇒ 2
(file-nlinks "doesnt-exist")
⇒ nil
```

**file-attributes** *filename* &optional *id-format* [Function]

This function returns a list of attributes of file *filename*. If the specified file cannot be opened, it returns `nil`. 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.

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.
1. The number of names the file has. Alternate names, also known as hard links, can be created by using the `add-name-to-file` function (see Section 25.7 [Changing Files], page 450).
2. The file's UID, normally as a string. However, if it does not correspond to a named user, the value is an integer or a floating point number.
3. The file's GID, likewise.
4. The time of last access, as a list of two integers. The first integer has the high-order 16 bits of time, the second has the low 16 bits. (This is similar to the value of `current-time`; see Section 39.5 [Time of Day], page 824.)
5. The time of last modification as a list of two integers (as above).
6. The time of last status change as a list of two integers (as above).
7. The size of the file in bytes. If the size is too large to fit in a Lisp integer, this is a floating point number.
8. The file's modes, as a string of ten letters or dashes, as in '`ls -l`'.
9. `t` if the file's GID would change if file were deleted and recreated; `nil` otherwise.
10. The file's inode number. If possible, this is an integer. If the inode number is too large to be represented as an integer in Emacs Lisp, then the value has the form `(high . low)`, where `low` holds the low 16 bits.
11. The file system number of the file system that the file is in. Depending on the magnitude of the value, this can be either an integer or a cons cell, in the same manner as the inode number. This element and the file's inode number together

give enough information to distinguish any two files on the system—no two files can have the same values for both of these numbers.

For example, here are the file attributes for ‘`files.texi`’:

```
(file-attributes "files.texi" 'string)
⇒ (nil 1 "lh" "users"
      (8489 20284)
      (8489 20284)
      (8489 20285)
      14906 "-rw-rw-rw-"
      nil 129500 -32252)
```

and here is how the result is interpreted:

<code>nil</code>	is neither a directory nor a symbolic link.
<code>1</code>	has only one name (the name ‘ <code>files.texi</code> ’ in the current default directory).
<code>"lh"</code>	is owned by the user with name “lh”.
<code>"users"</code>	is in the group with name “users”.
<code>(8489 20284)</code>	was last accessed on Aug 19 00:09.
<code>(8489 20284)</code>	was last modified on Aug 19 00:09.
<code>(8489 20285)</code>	last had its inode changed on Aug 19 00:09.
<code>14906</code>	is 14906 bytes long. (It may not contain 14906 characters, though, if some of the bytes belong to multibyte sequences.)
<code>"-rw-rw-rw-"</code>	has a mode of read and write access for the owner, group, and world.
<code>nil</code>	would retain the same GID if it were recreated.
<code>129500</code>	has an inode number of 129500.
<code>-32252</code>	is on file system number -32252.

### 25.6.5 How to Locate Files in Standard Places

This section explains how to search for a file in a list of directories (a *path*). One example is when you need to look for a program’s executable file, e.g., to find out whether a given program is installed on the user’s system. Another example is the search for Lisp libraries (see Section 15.3 [Library Search], page 203). Such searches generally need to try various possible file name extensions, in addition to various possible directories. Emacs provides a function for such a generalized search for a file.

**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 full *absolute*

*file name* of the file (see Section 25.8.2 [Relative File Names], page 455); 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 37.1 [Subprocess Creation], page 705), `load-suffixes`, `load-file-rep-suffixes` and the return value of the function `get-load-suffixes` (see Section 15.2 [Load Suffixes], page 203).

Typical values for *path* are `exec-path` (see Section 37.1 [Subprocess Creation], page 705) when looking for executable programs or `load-path` (see Section 15.3 [Library Search], page 203) 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 the predicate function to use for testing whether a candidate file is suitable. The predicate function is passed the candidate file name as its single argument. If *predicate* is `nil` or unspecified, `locate-file` uses `file-readable-p` as the default predicate. Useful non-default predicates include `file-executable-p`, `file-directory-p`, and other predicates described in Section 25.6.2 [Kinds of Files], page 445.

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`

[Function]

This function searches for the executable file of the named *program* and returns the full absolute name of the executable, including its file-name extensions, if any. It returns `nil` if the file is not found. The functions searches in all the directories in `exec-path` and tries all the file-name extensions in `exec-suffixes`.

## 25.7 Changing File Names and Attributes

The functions in this section rename, copy, delete, link, and set the modes of files.

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.

The next four commands all recursively follow symbolic links at all levels of parent directories for their first argument, but, if that argument is itself a symbolic link, then only `copy-file` replaces it with its (recursive) target.

#### `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*.

In the first part of the following example, we list two files, ‘foo’ and ‘foo3’.

```
% ls -li fo*
81908 -rw-rw-rw- 1 rms          29 Aug 18 20:32 foo
84302 -rw-rw-rw- 1 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      29 Aug 18 20:32 foo
81908 -rw-rw-rw- 2 rms      29 Aug 18 20:32 foo2
84302 -rw-rw-rw- 1 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      29 Aug 18 20:32 foo
81908 -rw-rw-rw- 3 rms      29 Aug 18 20:32 foo2
81908 -rw-rw-rw- 3 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 25.6.4 [File Attributes], page 447.

**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.

**copy-file** *oldname newname* &optional *ok-if-exists time* [Command]
*preserve-uid-gid*

This command copies the file *oldname* to *newname*. An error is signaled if *oldname* does not exist. If *newname* names a directory, it copies *oldname* into that directory, preserving its final name component.

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*.

This function copies the file modes, too.

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.

**make-symbolic-link** *filename newname &optional ok-if-exists* [Command]

This command makes a symbolic link to *filename*, named *newname*. This is like the shell command ‘`ln -s filename newname`’.

This function is not available on systems that don’t support symbolic links.

**delete-file** *filename* [Command]

This command deletes the file *filename*, like the shell command ‘`rm filename`’. If the file has multiple names, it continues to exist under the other names.

A suitable kind of **file-error** error is signaled if the file does not exist, or is not deletable. (On Unix and GNU/Linux, a file is deletable if its directory is writable.)

If *filename* is a symbolic link, **delete-file** does not replace it with its target, but it does follow symbolic links at all levels of parent directories.

See also **delete-directory** in Section 25.10 [Create/Delete Dirs], page 464.

**define-logical-name** *varname string* [Function]

This function defines the logical name *varname* to have the value *string*. It is available only on VMS.

**set-file-modes** *filename mode* [Function]

This function sets mode bits of *filename* to *mode* (which must be an integer). Only the low 12 bits of *mode* are used. This function recursively follows symbolic links at all levels for *filename*.

**set-default-file-modes** *mode* [Function]

This function sets the default file protection for new files created by Emacs and its subprocesses. Every file created with Emacs initially has this protection, or a subset of it (**write-region** will not give a file execute permission even if the default file protection allows execute permission). On Unix and GNU/Linux, the default protection is the bitwise complement of the “umask” value.

The argument *mode* must be an integer. On most systems, only the low 9 bits of *mode* are meaningful. You can use the Lisp construct for octal character codes to enter *mode*; for example,

```
(set-default-file-modes ?\644)
```

Saving a modified version of an existing file does not count as creating the file; it preserves the existing file’s mode, whatever that is. So the default file protection has no effect.

**default-file-modes** [Function]

This function returns the current default protection value.

**set-file-times** *filename &optional time* [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 in the format returned by **current-time** (see Section 39.5 [Time of Day], page 824).

On MS-DOS, there is no such thing as an “executable” file mode bit. So Emacs 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 Unix-standard ‘#!’ signature, such as shell and Perl scripts, are also considered as executable files. This is reflected in the values returned by `file-modes` and `file-attributes`. Directories are also reported with executable bit set, for compatibility with Unix.

## 25.8 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 Unix syntax; but they always return Unix syntax. On VMS, these functions (and the ones that operate on files) understand both VMS file-name syntax and Unix syntax. This enables Lisp programs to specify file names in Unix syntax and work properly on all systems without change.

### 25.8.1 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.

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. The rules in VMS syntax are complicated.

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. On VMS, every file has a version number, but most of the time the file name actually used in Emacs omits the version number, so that version numbers in Emacs are found mostly in directory lists.

**`file-name-directory filename`** [Function]

This function returns the directory part of *filename*, as a directory name (see Section 25.8.3 [Directory Names], page 456), or `nil` if *filename* does not include a directory part.

On GNU and Unix systems, a string returned by this function always ends in a slash. On MS-DOS it can also end in a colon. On VMS, it returns a string ending in one of the three characters ‘:’, ‘]’, or ‘>’.

```
(file-name-directory "lewis/foo") ; Unix example
⇒ "lewis/"
(file-name-directory "foo")      ; Unix example
⇒ nil
(file-name-directory "[X]FOO.TMP") ; VMS example
⇒ "[X]"
```

**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/")
⇒ ""
;; The following example is accurate only on VMS.
(file-name-nondirectory "[X]FOO.TMP")
⇒ "FOO.TMP"
```

**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"
;; The following example applies to VMS only.
(file-name-sans-versions "foo;23")
⇒ "foo"
```

**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 starts with 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-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.

### 25.8.2 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 Unix and GNU/Linux, an absolute file name starts with a slash or a tilde (~), 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*. The rules on VMS are complicated.

**file-name-absolute-p** *filename* [Function]

This function returns t if file *filename* is an absolute file name, nil otherwise. On VMS, this function understands both Unix syntax and VMS syntax.

```
(file-name-absolute-p "~rms/foo")
  ⇒ t
(file-name-absolute-p "rms/foo")
  ⇒ nil
(file-name-absolute-p "/user/rms/foo")
  ⇒ t
```

Given a possibly relative file name, you can convert it to an absolute name using `expand-file-name` (see Section 25.8.4 [File Name Expansion], page 457). 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"
```

### 25.8.3 Directory Names

A *directory name* is the name of a directory. A directory is actually a kind of file, so it has a file name, which is related to the directory name but not identical to it. (This is not quite the same as the usual Unix terminology.) These two different names for the same entity are related by a syntactic transformation. On GNU and Unix systems, this is simple: a directory name ends in a slash, whereas the directory's name as a file lacks that slash. On MS-DOS and VMS, the relationship is more complicated.

The difference between a directory name and its name as a file is subtle but crucial. When an Emacs variable or function argument is described as being a directory name, a file name of a directory 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 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. On most systems, this means appending a slash to the string (if it does not already end in one). On VMS, the function converts a string of the form '`[X]Y.DIR.1`' to the form '`[X.Y]`'.

```
(file-name-as-directory "~rms/lewis")
  ⇒ "~rms/lewis/"
```

**`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. On most systems, this means removing the final slash (or backslash) from the string. On VMS, the function converts a string of the form '`[X.Y]`' to '`[X]Y.DIR.1`'.

```
(directory-file-name "~lewis/")
  ⇒ "~lewis"
```

Given a directory name, you can combine it with a relative file name using `concat`:

```
(concat dirname relfile)
```

Be sure to verify that the file name is relative before doing that. If you use an absolute file name, the results could be syntactically invalid or refer to the wrong file.

If you want to use a directory file name in making such a combination, you must first convert it to a directory name using `file-name-as-directory`:

```
(concat (file-name-as-directory dirfile) relfile)
```

Don't try concatenating a slash by hand, as in

```
;;; Wrong!
(concat dirfile "/" relfile)
```

because this is not portable. Always use **file-name-as-directory**.

Directory name abbreviations are useful for directories that are normally accessed through symbolic links. Sometimes the users recognize primarily the link's name as “the name” of the directory, and find it annoying to see the directory's “real” name. If you define the link name as an abbreviation for the “real” name, Emacs shows users the abbreviation instead.

#### **directory-abbrev-alist**

[Variable]

The variable **directory-abbrev-alist** contains an alist of abbreviations to use for file directories. Each element has the form (**from . to**), and says to replace *from* with *to* when it appears in a directory name. The *from* string is actually a regular expression; it should always start with ‘~’. The *to* string should be an ordinary absolute directory name. Do not use ‘~’ to stand for a home directory in that string. The function **abbreviate-file-name** performs these substitutions.

You can set this variable in ‘`site-init.el`’ to describe the abbreviations appropriate for your site.

Here's an example, from a system on which file system ‘`/home/fsf`’ and so on are normally accessed through symbolic links named ‘`/fsf`’ and so on.

```
((("~/home/fsf" . "/fsf"))
 ("~/home/gp" . "/gp")
 ("~/home/gd" . "/gd"))
```

To convert a directory name to its abbreviation, use this function:

#### **abbreviate-file-name filename**

[Function]

This function applies abbreviations from **directory-abbrev-alist** to its argument, and substitutes ‘~’ for the user's home directory. You can use it for directory names and for file names, because it recognizes abbreviations even as part of the name.

### 25.8.4 Functions that Expand Filenames

*Expansion* of 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 name as well as the file name to be expanded. Expansion also simplifies file names by eliminating redundancies such as ‘`./`’ 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. (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"
(expand-file-name "$HOME/foo")
  ⇒ "/xcssun/users/rms/lewis/$HOME/foo"
```

If the part of the combined file name before the first slash is ‘~’, it expands to the value of the `HOME` environment variable (usually your home directory). If the part before the first slash is ‘~*user*’ and if *user* is a valid login name, it expands to *user*’s home directory.

Filenames 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 ‘/’.

Note that `expand-file-name` does *not* expand environment variables; only `substitute-in-file-name` does that.

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 25.6.3 [Truenames], page 446.

### `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`.

Aside from VMS, 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 name, 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.
```

On VMS, '\$' substitution is not done, so this function does nothing on VMS except discard superfluous initial components as shown above.

### 25.8.5 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* [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 empty file. On MS-DOS, this function can truncate the *string* *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 25.8.3 [Directory Names], page 456.

If *suffix* is non-nil, `make-temp-file` adds it at the end of the file name.

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`.

In older Emacs versions where `make-temp-file` does not exist, you should use `make-temp-name` instead:

```
(make-temp-name
  (expand-file-name name-of-application
    temporary-file-directory))
```

#### `make-temp-name string`

[Function]

This function generates a string that can be used as a unique file name. The name starts with *string*, and has several random characters appended to it, which are different in each Emacs job. It is like `make-temp-file` except that it just constructs a name, and does not create a file. Another difference is that *string* should be an absolute file name. On MS-DOS, this function can truncate the *string* prefix to fit into the 8+3 file-name limits.

#### `temporary-file-directory`

[Variable]

This variable specifies the directory name for creating temporary files. Its value should be a directory name (see Section 25.8.3 [Directory Names], page 456), 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`

[Variable]

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)))
```

### 25.8.6 File Name Completion

This section describes low-level subroutines for completing a file name. For higher level functions, see Section 20.6.5 [Reading File Names], page 293.

**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`’.

### 25.8.7 Standard File Names

Most of the file names used in Lisp programs are entered by the user. But occasionally a Lisp program needs to specify a standard file name for a particular use—typically, to hold customization information about each user. For example, abbrev definitions are stored (by default) in the file ‘`~/.abbrev_defs`’; the `completion` package stores completions in the file ‘`~/.completions`’. These are two of the many standard file names used by parts of Emacs for certain purposes.

Various operating systems have their own conventions for valid file names and for which file names to use for user profile data. A Lisp program which reads a file using a standard file name ought to use, on each type of system, a file name suitable for that system. The function `convert-standard-filename` makes this easy to do.

**convert-standard-filename** *filename* [Function]

This function alters the file name *filename* to fit the conventions of the operating system in use, and returns the result as a new string.

The recommended way to specify a standard file name in a Lisp program is to choose a name which fits the conventions of GNU and Unix systems, usually with a nondirectory part that starts with a period, and pass it to `convert-standard-filename` instead of using it directly. Here is an example from the `completion` package:

```
(defvar save-completions-file-name
  (convert-standard-filename " ~/.completions")
  "*The file name to save completions to.")
```

On GNU and Unix systems, and on some other systems as well, `convert-standard-filename` returns its argument unchanged. On some other systems, it alters the name to fit the system’s conventions.

For example, on MS-DOS the alterations made by this function include converting a leading ‘.’ to ‘\_’, converting a ‘\_’ in the middle of the name to ‘.’ if there is no other ‘.’, inserting a ‘.’ after eight characters if there is none, and truncating to three characters after the ‘..’. (It makes other changes as well.) Thus, ‘`.abbrev_defs`’ becomes ‘`_abbrev.def`’, and ‘`.completions`’ becomes ‘`_complet.ion`’.

## 25.9 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* [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 that 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.

```
(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-files-and-attributes** *directory* &optional *full-name* [Function]
*match-regexp* *nosort* *id-format*

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** would return 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 448).

**file-name-all-versions** *file* *dirname* [Function]

This function returns a list of all versions of the file named *file* in directory *dirname*. It is only available on VMS.

**file-expand-wildcards** *pattern* &optional *full* [Function]

This function expands the wildcard pattern *pattern*, returning a list of file names that match it.

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 name 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.

## 25.10 Creating 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* [Function]

This function 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.

**delete-directory** *dirname* [Function]

This function 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 the directory contains any files, `delete-directory` signals an error.

This function only follows symbolic links at the level of parent directories.

## 25.11 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 remote file names (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 do 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:

```
access-file, add-name-to-file,  
byte-compiler-base-file-name,  
copy-file, delete-directory,  
delete-file,  
diff-latest-backup-file,  
directory-file-name,  
directory-files,  
directory-files-and-attributes,  
dired-call-process,  
dired-compress-file, dired-uncache,  
expand-file-name,  
file-accessible-directory-p,  
file-attributes,  
file-directory-p,  
file-executable-p, file-exists-p,  
file-local-copy, file-remote-p,  
file-modes, file-name-all-completions,  
file-name-as-directory,  
file-name-completion,  
file-name-directory,  
file-name-nondirectory,  
file-name-sans VERSIONS, file-newer-than-file-p,  
file-ownership-preserved-p,
```

```
file-readable-p, file-regular-p, file-symlink-p,
file-truename, file-writable-p,
find-backup-file-name,
find-file-noselect,
get-file-buffer,
insert-directory,
insert-file-contents,
load, make-directory,
make-directory-internal,
make-symbolic-link,
rename-file, set-file-modes,
set-visited-file-modtime, shell-command,
substitute-in-file-name,
unhandled-file-name-directory,
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***

[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 filename `/ssh:user@host:/some/file` is `/ssh:user@host::`.

If `file-remote-p` returns the same identifier for two different filenames, 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. Implementors of file handlers need to ensure this principle is valid.

**unhandled-file-name-directory** *filename* [Function]

This function returns the name of a directory that is not magic. It uses the directory part of *filename* if that is not magic. For a magic file name, it invokes the file name handler, which therefore decides what value to return.

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.

## 25.12 File Format Conversion

The variable **format-alist** defines a list of *file formats*, which describe textual representations used in files for the data (text, text-properties, and possibly other information) in an Emacs buffer. Emacs performs format conversion if appropriate when reading and writing files.

**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)*

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.

*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.

*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.

`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.

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 format `format`, and makes that format the default for future saves of the buffer. The argument `format` is a list of format names. 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 438.

#### `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 25.3 [Reading from Files], page 440).

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 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.

### 26.1 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.

#### 26.1.1 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 . backupname)`, where *modes* are the mode bits of the original file, as returned by `file-modes` (see Section 25.6.4 [Other Information about Files], page 447), and *backupname* is the name of the backup. In all other cases, that is, 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
          (function (lambda ()
                      (make-local-variable
                        'make-backup-files)
                      (setq 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** [Variable]

This variable's value is an alist of filename patterns and backup directory names. 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 name.

If this variable is `nil`, 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** [Variable]

This variable's value is a function to use for making backups instead of the default `make-backup-file-name`. A value of `nil` gives the default `make-backup-file-name` behavior. See Section 26.1.4 [Naming Backup Files], page 474.

This could be buffer-local to do something special for specific files. If you define it, you may need to change `backup-file-name-p` and `file-name-sans VERSIONS` too.

### 26.1.2 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 25.2 [Saving Buffers], page 437.

#### `backup-by-copying`

[User Option]

If this variable is non-`nil`, Emacs always makes backup files by copying.

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).

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`, 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 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 only, when necessary to prevent a change in the owner of the file.

The default is 200.

### 26.1.3 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 26.1.4 [Backup Names], page 474) 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.

### 26.1.4 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 26.1.3 [Numbered Backups], page 474.

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-newest-backup *filename*** [Function]

This function returns the name of the most recent backup file for *filename*, or `nil` if that file has no backup files.

Some file comparison commands use this function so that they can automatically compare a file with its most recent backup.

## 26.2 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]

When used interactively without an argument, this command is a toggle switch: it turns on auto-saving of the current buffer if it is off, and vice versa. With an argument *arg*, the command turns auto-saving on if the value of *arg* is `t`, a nonempty list, or a positive integer. Otherwise, it turns auto-saving off.

**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
```

The standard definition of this function is as follows:

```
(defun auto-save-file-name-p (filename)
  "Return non-nil if FILENAME can be yielded by..."
  (string-match "^#.*#$" filename))
```

This function exists so that you can customize it if you wish to change the naming convention for auto-save files. If you redefine it, be sure to redefine the function `make-auto-save-file-name` correspondingly.

**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#"
```

Here is a simplified version of the standard definition of this function:

```
(defun make-auto-save-file-name ()
  "Return file name to use for auto-saves \
  of current buffer.."
  (if buffer-file-name
      (concat
       (file-name-directory buffer-file-name)
       "#"
       (file-name-nondirectory buffer-file-name)
       "#")
      (expand-file-name
       (concat "#%" (buffer-name) "#"))))
```

This exists as a separate function so that you can redefine it to customize the naming convention for auto-save files. Be sure to change `auto-save-file-name-p` in a corresponding way.

**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.

**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.

**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** [Variable]

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**.

## 26.3 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 problematical.

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. If non-**nil**, 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. If the value is **nil**, reverting works the usual way.

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 function to regenerate the contents.

**revert-buffer-insert-file-contents-function**

[Variable]

The value of this variable, if non-**nil**, 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 set 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 **revert-buffer** before inserting the modified contents—but only if **revert-buffer-function** is **nil**.

**after-revert-hook**

[Variable]

This normal hook is run by **revert-buffer** after inserting the modified contents—but only if **revert-buffer-function** is **nil**.

## 27 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.

### 27.1 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 32 [Text], page 581.

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 11.10 [Buffer-Local Variables], page 147.

For functions and variables related to visiting files in buffers, see Section 25.1 [Visiting Files], page 434 and Section 25.2 [Saving Buffers], page 437. For functions and variables related to the display of buffers in windows, see Section 28.6 [Buffers and Windows], page 505.

**bufferp** *object* [Function]

This function returns `t` if *object* is a buffer, `nil` otherwise.

### 27.2 The Current Buffer

There are, in general, many buffers in an Emacs session. At any time, one of them is designated as the *current buffer*. This is the buffer in which most editing takes place, because most of the primitives for examining or changing text in a buffer operate implicitly on the current buffer (see Chapter 32 [Text], page 581). Normally the buffer that is displayed on the screen in the selected window 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 way to designate a current buffer in a Lisp program is by calling `set-buffer`. The specified buffer remains current until a new one is designated.

When an editing command returns to the editor command loop, the command loop designates the buffer displayed in the selected window as current, to prevent confusion: the buffer that the cursor is in when Emacs reads a command is the buffer that the command will apply to. (See Chapter 21 [Command Loop], page 304.) Therefore, `set-buffer` is not

the way to switch visibly to a different buffer so that the user can edit it. For that, you must use the functions described in Section 28.7 [Displaying Buffers], page 506.

**Warning:** Lisp functions that change to a different current buffer should not depend on the command loop to set it back afterwards. Editing commands written in Emacs Lisp can be called from other programs as well as 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). Therefore, you should normally use `set-buffer` within a `save-current-buffer` or `save-excursion` (see Section 30.3 [Excursions], page 568) form that will restore the current buffer when your function is done. Here is an example, the code for the command `append-to-buffer` (with the documentation string abridged):

```
(defun append-to-buffer (buffer start end)
  "Append to specified buffer the text of the region.
...
(interactive \"BAppend to buffer: \\nr\")
(let ((oldbuf (current-buffer)))
  (save-current-buffer
    (set-buffer (get-buffer-create buffer))
    (insert-buffer-substring oldbuf start end))))
```

This function binds a local variable to record the current buffer, and then `save-current-buffer` arranges to make it current again. Next, `set-buffer` makes the specified buffer current. Finally, `insert-buffer-substring` copies the string from the original current buffer to the specified (and now current) buffer.

If the buffer appended to happens to be displayed in some window, the next redisplay will show how its text has changed. Otherwise, you will not see the change immediately on the screen. The buffer becomes current temporarily during the execution of the command, but this 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! There are two ways to do this. In simple cases, you may see that nothing ever changes the current buffer within the scope of the binding. Otherwise, use `save-current-buffer` or `save-excursion` to make sure that the buffer current at the beginning is current again whenever the variable is unbound.

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. Here is what *not* to do:

```
(let (buffer-read-only
      (obuf (current-buffer)))
  (set-buffer ...)
  ...
  (set-buffer obuf))
```

Using `save-current-buffer`, as shown here, handles quitting, errors, and `throw`, as well as ordinary evaluation.

```
(let (buffer-read-only)
  (save-current-buffer
    (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. This does not display the buffer in any window, so the user cannot necessarily see the buffer. But Lisp programs will now operate on it.

This function returns the buffer identified by *buffer-or-name*. An error is signaled if *buffer-or-name* does not identify an existing buffer.

**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 10.5 [Nonlocal Exits], page 125).

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 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 10.5 [Nonlocal Exits], page 125).

An error is signaled if *buffer-or-name* does not identify an existing buffer.

**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 32.9 [Undo], page 596) is not recorded in the buffer created by this macro (but *body* can enable that, if needed).

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 10.5 [Nonlocal Exits], page 125).

See also **with-temp-file** in [Writing to Files], page 442.

## 27.3 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 32.9 [Undo], page 596.

**buffer-name** &optional *buffer* [Function]

This function returns the name of *buffer* as a string. If *buffer* is not supplied, it defaults to the current buffer.

If `buffer-name` returns `nil`, it means that *buffer* has been killed. See Section 27.10 [Killing Buffers], page 493.

```
(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 27.9 [Creating Buffers], page 492.

`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 27.9 [Creating Buffers], page 492.

## 27.4 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 25.1 [Visiting Files], page 434.

`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 25.6.3 [Truenames], page 446, and [Definition of abbreviate-file-name], page 457.

**buffer-file-number** [Variable]

This buffer-local variable holds the file number and directory device number 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 (`filenum devnum`). This pair of numbers uniquely identifies the file among all files accessible on the system. See the function `file-attributes`, in Section 25.6.4 [File Attributes], page 447, for more information about them.

If `buffer-file-name` is the name of a symbolic link, then both numbers refer to the recursive target.

**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 25.8.4 [File Name Expansion], page 457), 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 *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 27.6 [Modification Time], page 488). 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.

## 27.5 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 23.4.4 [Mode Line Variables], page 405), and controls saving (see Section 25.2 [Saving Buffers], page 437) and auto-saving (see Section 26.2 [Auto-Saving], page 476).

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 32 [Text], page 581.

#### **buffer-modified-p &optional buffer**

[Function]

This function returns *t* if the buffer *buffer* has been modified since it was last read in from a file or saved, or *nil* otherwise. If *buffer* is not supplied, the current buffer is tested.

#### **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.

**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. The counter can wrap around occasionally.

**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.

## 27.6 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 25.6.4 [File Attributes], page 447, for how to examine a file's modification time.)

**verify-visited-file-modtime buffer** [Function]

This function compares what *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 list of the form `(high low)`. (This is the same format that `file-attributes` uses to return time values; see Section 25.6.4 [File Attributes], page 447.)

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 list 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`.

For a new buffer visiting a not yet existing file, `high` is `-1` and `low` is `65535`, that is,  $2^{16} - 1$ .

**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 zero, it should have the form `(high . low)` or `(high low)`, in either case containing two integers, each of which holds 16 bits of the time.

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 an 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 of the buffer. 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 25.5 [File Locks], page 442.

## 27.7 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`.

**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 non-`nil` `read-only` properties (either text properties or overlay properties). See Section 32.19.4 [Special Properties], page 620, for more information about text properties. See Section 38.9 [Overlays], page 754, for more information about overlays and their 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`).

**toggle-read-only &optional arg**

[Command]

This command toggles whether the current buffer is read-only. It is intended for interactive use; do not use it in programs. At any given point in a program, you should know whether you want the read-only flag on or off; so you can set `buffer-read-only` explicitly to the proper value, `t` or `nil`.

If `arg` is non-`nil`, it should be a raw prefix argument. `toggle-read-only` sets `buffer-read-only` to `t` if the numeric value of that prefix argument is positive and to `nil` otherwise. See Section 21.11 [Prefix Command Arguments], page 340.

**barf-if-buffer-read-only**

[Function]

This function signals a `buffer-read-only` error if the current buffer is read-only. See Section 21.2.1 [Using Interactive], page 305, for another way to signal an error if the current buffer is read-only.

## 27.8 The Buffer List

The *buffer list* is a list of all live buffers. The order of the buffers in the 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. Buffers move to the front of the list when they are selected for display in a window (see Section 28.7 [Displaying Buffers], page 506), and to the end when they are 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 Emacs buffer list, each frame has its own version of the buffer list, in which the buffers that have been selected in that frame come first, starting with

the buffers most recently selected *in that frame*. (This order is recorded in *frame*'s **buffer-list** frame parameter; see Section 29.3.3.5 [Buffer Parameters], page 535.) The buffers that were never selected in *frame* come afterward, ordered according to the fundamental Emacs 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 buffer list. If *frame* is **nil**, the fundamental Emacs buffer list is used: all the buffers appear in order of most recent selection, regardless of which frames they were selected in.

```
(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 (function buffer-name) (buffer-list))
  ⇒ ("buffers.texi" " *Minibuf-1*"
      "buffer.c" "*Help*" "TAGS")
```

The list that **buffer-list** returns is constructed specifically by **buffer-list**; 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 frame-independent 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 frame's buffer list, set the frame's **buffer-list** frame parameter with **modify-frame-parameters** (see Section 29.3.1 [Parameter Access], page 531).

**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 selected most recently (in frame *frame* or else the currently selected frame, see Section 29.9 [Input Focus], page 543), 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 buffer), then **other-buffer** returns the first buffer in the selected frame's buffer list that is not now visible in any window in a visible frame.

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 29.3.3.5 [Buffer Parameters], page 535.

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).

### `bury-buffer` &optional *buffer-or-name*

[Command]

This function 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.

`bury-buffer` operates on each frame’s `buffer-list` parameter as well as the frame-independent Emacs 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 nil)`.

If *buffer-or-name* is `nil` or omitted, this means to bury the current buffer. In addition, if the buffer is displayed in the selected window, this switches to some other buffer (obtained using `other-buffer`) in the selected window. But if the buffer 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 28.6 [Buffers and Windows], page 505.

## 27.9 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.

Other functions you can use to create buffers include `with-output-to-temp-buffer` (see Section 38.8 [Temporary Displays], page 752) and `create-file-buffer` (see Section 25.1 [Visiting Files], page 434). Starting a subprocess can also create a buffer (see Chapter 37 [Processes], page 705).

### `get-buffer-create` *name*

[Function]

This function returns a buffer named *name*. It returns a live buffer with that name, if one exists; otherwise, it creates a new buffer. The buffer does not become the current buffer—this function does not change which buffer is current.

If *name* is a buffer instead of a string, it is returned, even if it is dead. An error is signaled if *name* is neither a string nor a buffer.

```
(get-buffer-create "foo")
⇒ #<buffer foo>
```

The major mode for a newly created buffer is set to Fundamental mode. (The variable `default-major-mode` is handled at a higher level; see Section 23.2.3 [Auto Major Mode], page 388.) If the name begins with a space, the buffer initially disables undo information recording (see Section 32.9 [Undo], page 596).

**generate-new-buffer name** [Function]

This function returns a newly created, empty buffer, but does not make it current. If there is no buffer named *name*, then that is the name of the new buffer. If that name is in use, this function adds suffixes of the form ‘<*n*>’ to *name*, where *n* is an integer. It tries successive integers starting with 2 until it finds an available 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 variable `default-major-mode` is handled at a higher level. See Section 23.2.3 [Auto Major Mode], page 388.

See the related function `generate-new-buffer-name` in Section 27.3 [Buffer Names], page 484.

## 27.10 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 in general 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 27.2 [Current Buffer], page 481.

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 killed buffer is `nil`. You can use this feature to test whether a buffer has been killed:

```
(defun buffer-killed-p (buffer)
  "Return t if BUFFER is killed."
  (not (buffer-name buffer)))
```

**kill-buffer 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`, it kills the current buffer.

Any processes that have this buffer as the `process-buffer` are sent the `SIGHUP` signal, which normally causes them to terminate. (The basic meaning of `SIGHUP` is that a dialup line has been disconnected.) See Section 37.8 [Signals to Processes], page 715.

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 27.5 [Buffer Modification], page 487.

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]

After 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.

#### `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 23.1 [Hooks], page 382. This variable is a permanent local, so its local binding is not cleared by changing major modes.

#### `buffer-offer-save`

[Variable]

This variable, if non-`nil` in a particular buffer, tells `save-buffers-kill-emacs` and `save-some-buffers` (if the second optional argument to that function is `t`) to offer to save that buffer, just as they offer to save file-visiting buffers. See [Definition of `save-some-buffers`], page 438. The variable `buffer-offer-save` automatically becomes buffer-local when set for any reason. See Section 11.10 [Buffer-Local Variables], page 147.

#### `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 buffer which has not been killed, `nil` otherwise.

## 27.11 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* [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*.

**clone-indirect-buffer** *newname display-flag &optional norecord* [Function]

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*, 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.

## 27.12 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.

## 28 Windows

This chapter describes most of the functions and variables related to Emacs windows. See Chapter 38 [Display], page 739, for information on how text is displayed in windows.

### 28.1 Basic Concepts of Emacs Windows

A *window* in Emacs is the physical area of the screen in which a buffer is displayed. The term is also used to refer to a Lisp object that represents that screen area in Emacs Lisp. It should be clear from the context which is meant.

Emacs groups windows into frames. A frame represents an area of screen available for Emacs to use. Each frame always contains at least one window, but you can subdivide it vertically or horizontally into multiple nonoverlapping Emacs windows.

In each frame, at any time, one and only one window is designated as *selected within the frame*. The frame's cursor appears in that window, but the other windows have “non-selected” cursors, normally less visible. At any time, one frame is the selected frame; and the window selected within that frame is *the selected window*. The selected window's buffer is usually the current buffer (except when `set-buffer` has been used). See Section 27.2 [Current Buffer], page 481.

**cursor-in-non-selected-windows**

[Variable]

If this variable is `nil`, Emacs displays only one cursor, in the selected window. Other windows have no cursor at all.

For practical purposes, a window exists only while it is displayed in a frame. Once removed from the frame, the window is effectively deleted and should not be used, *even though there may still be references to it* from other Lisp objects. Restoring a saved window configuration is the only way for a window no longer on the screen to come back to life. (See Section 28.3 [Deleting Windows], page 501.)

Each window has the following attributes:

- containing frame
- window height
- window width
- window edges with respect to the screen or frame
- the buffer it displays
- position within the buffer at the upper left of the window
- amount of horizontal scrolling, in columns
- point
- the mark
- how recently the window was selected
- fringe settings
- display margins
- scroll-bar settings

Users create multiple windows so they can look at several buffers at once. Lisp libraries use multiple windows for a variety of reasons, but most often to display related information. In Rmail, for example, you can move through a summary buffer in one window while the other window shows messages one at a time as they are reached.

The meaning of “window” in Emacs is similar to what it means in the context of general-purpose window systems such as X, but not identical. The X Window System places X windows on the screen; Emacs uses one or more X windows as frames, and subdivides them into Emacs windows. When you use Emacs on a character-only terminal, Emacs treats the whole terminal screen as one frame.

Most window systems support arbitrarily located overlapping windows. In contrast, Emacs windows are *tiled*; they never overlap, and together they fill the whole screen or frame. Because of the way in which Emacs creates new windows and resizes them, not all conceivable tilings of windows on an Emacs frame are actually possible. See Section 28.2 [Splitting Windows], page 498, and Section 28.14 [Size of Window], page 520.

See Chapter 38 [Display], page 739, for information on how the contents of the window’s buffer are displayed in the window.

**windowp** *object*

[Function]

This function returns `t` if *object* is a window.

## 28.2 Splitting Windows

The functions described here are the primitives used to split a window into two windows. Two higher level functions sometimes split a window, but not always: `pop-to-buffer` and `display-buffer` (see Section 28.7 [Displaying Buffers], page 506).

The functions described here do not accept a buffer as an argument. The two “halves” of the split window initially display the same buffer previously visible in the window that was split.

**split-window &optional** *window size horizontal*

[Command]

This function splits a new window out of *window*’s screen area. It returns the new window.

If *horizontal* is non-`nil`, then *window* splits into two side by side windows. The original window *window* keeps the leftmost *size* columns, and gives the rest of the columns to the new window. Otherwise, it splits into windows one above the other, and *window* keeps the upper *size* lines and gives the rest of the lines to the new window. The original window is therefore the left-hand or upper of the two, and the new window is the right-hand or lower.

If *window* is omitted or `nil`, that stands for the selected window. When you split the selected window, it remains selected.

If *size* is omitted or `nil`, then *window* is divided evenly into two parts. (If there is an odd line, it is allocated to the new window.) When `split-window` is called interactively, all its arguments are `nil`.

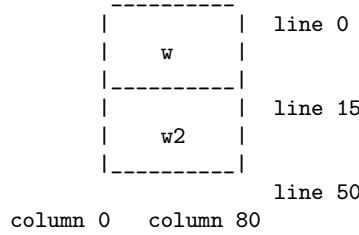
If splitting would result in making a window that is smaller than `window-min-height` or `window-min-width`, the function signals an error and does not split the window at all.

The following example starts with one window on a screen that is 50 lines high by 80 columns wide; then it splits the window.

```
(setq w (selected-window))
⇒ #<window 8 on windows.texi>
(window-edges)           ; Edges in order:
⇒ (0 0 80 50)           ; left-top-right-bottom

;; Returns window created
(setq w2 (split-window w 15))
⇒ #<window 28 on windows.texi>
(window-edges w2)
⇒ (0 15 80 50)           ; Bottom window;
                           ; top is line 15
(window-edges w)
⇒ (0 0 80 15)            ; Top window
```

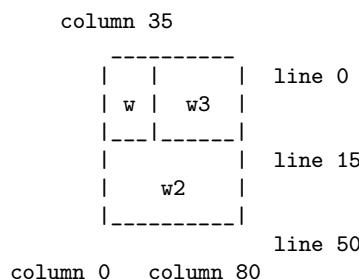
The screen looks like this:



Next, split the top window horizontally:

```
(setq w3 (split-window w 35 t))
⇒ #<window 32 on windows.texi>
(window-edges w3)
⇒ (35 0 80 15)           ; Left edge at column 35
(window-edges w)
⇒ (0 0 35 15)            ; Right edge at column 35
(window-edges w2)
⇒ (0 15 80 50)            ; Bottom window unchanged
```

Now the screen looks like this:



Normally, Emacs indicates the border between two side-by-side windows with a scroll bar (see Section 29.3.3.4 [Layout Parameters], page 534) or ‘|’ characters. The display table can specify alternative border characters; see Section 38.21 [Display Tables], page 807.

**split-window-vertically &optional size** [Command]

This function splits the selected window into two windows, one above the other, leaving the upper of the two windows selected, with *size* lines. (If *size* is negative, then the lower of the two windows gets  $-size$  lines and the upper window gets the

rest, but the upper window is still the one selected.) However, if `split-window-keep-point` (see below) is `nil`, then either window can be selected.

In other respects, this function is similar to `split-window`. In particular, the upper window is the original one and the return value is the new, lower window.

**split-window-keep-point**

[User Option]

If this variable is non-`nil` (the default), then `split-window-vertically` behaves as described above.

If it is `nil`, then `split-window-vertically` adjusts point in each of the two windows to avoid scrolling. (This is useful on slow terminals.) It selects whichever window contains the screen line that point was previously on.

This variable only affects the behavior of `split-window-vertically`. It has no effect on the other functions described here.

**split-window-horizontally &optional size**

[Command]

This function splits the selected window into two windows side-by-side, leaving the selected window on the left with `size` columns. If `size` is negative, the rightmost window gets  $-size$  columns, but the leftmost window still remains selected.

This function is basically an interface to `split-window`. You could define a simplified version of the function like this:

```
(defun split-window-horizontally (&optional arg)
  "Split selected window into two windows, side by side..."
  (interactive "P")
  (let ((size (and arg (prefix-numeric-value arg))))
    (and size (< size 0)
         (setq size (+ (window-width) size)))
    (split-window nil size t)))
```

**one-window-p &optional no-mini all-frames**

[Function]

This function returns non-`nil` if there is only one window. The argument `no-mini`, if non-`nil`, means don't count the minibuffer even if it is active; otherwise, the minibuffer window is counted when it is active.

The argument `all-frames` specifies which frames to consider. Here are the possible values and their meanings:

`nil` Count the windows in the selected frame, plus the minibuffer used by that frame even if it lies in some other frame.

`t` Count all windows in all existing frames.

`visible` Count all windows in all visible frames.

`0` Count all windows in all visible or iconified frames.

anything else

Count precisely the windows in the selected frame, and no others.

## 28.3 Deleting Windows

A window remains visible on its frame unless you *delete* it by calling certain functions that delete windows. A deleted window cannot appear on the screen, but continues to exist as a Lisp object until there are no references to it. There is no way to cancel the deletion of a window aside from restoring a saved window configuration (see Section 28.18 [Window Configurations], page 526). Restoring a window configuration also deletes any windows that aren't part of that configuration.

When you delete a window, the space it took up is given to one adjacent sibling.

**window-live-p** *window* [Function]

This function returns `nil` if *window* is deleted, and `t` otherwise.

**Warning:** Erroneous information or fatal errors may result from using a deleted window as if it were live.

**delete-window** &optional *window* [Command]

This function removes *window* from display, and returns `nil`. If *window* is omitted, then the selected window is deleted. An error is signaled if there is only one window when **delete-window** is called.

**delete-other-windows** &optional *window* [Command]

This function makes *window* the only window on its frame, by deleting the other windows in that frame. If *window* is omitted or `nil`, then the selected window is used by default.

The return value is `nil`.

**delete-windows-on** *buffer-or-name* &optional *frame* [Command]

This function deletes all windows showing *buffer-or-name*. If there are no windows showing *buffer-or-name*, it does nothing. *buffer-or-name* must be a buffer or the name of an existing buffer.

**delete-windows-on** operates frame by frame. If a frame has several windows showing different buffers, then those showing *buffer-or-name* are removed, and the others expand to fill the space. If all windows in some frame are showing *buffer-or-name* (including the case where there is only one window), then the frame winds up with a single window showing another buffer chosen with **other-buffer**. See Section 27.8 [The Buffer List], page 490.

The argument *frame* controls which frames to operate on. This function does not use it in quite the same way as the other functions which scan all windows; specifically, the values `t` and `nil` have the opposite of their meanings in other functions. Here are the full details:

- If it is `nil`, operate on all frames.
- If it is `t`, operate on the selected frame.
- If it is **visible**, operate on all visible frames.
- If it is `0`, operate on all visible or iconified frames.
- If it is a frame, operate on that frame.

This function always returns `nil`.

## 28.4 Selecting Windows

When a window is selected, the buffer in the window becomes the current buffer, and the cursor will appear in it.

**selected-window** [Function]

This function returns the selected window. This is the window in which the cursor appears and to which many commands apply.

**select-window window &optional norecord** [Function]

This function makes *window* the selected window. The cursor then appears in *window* (on redisplay). Unless *window* was already selected, **select-window** makes *window*'s buffer the current buffer.

Normally *window*'s selected buffer is moved to the front of the buffer list, but if *norecord* is non-*nil*, the buffer list order is unchanged.

The return value is *window*.

```
(setq w (next-window))
(select-window w)
⇒ #<window 65 on windows.texi>
```

**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 the *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.

**with-selected-window window forms...** [Macro]

This macro selects *window* (without changing the buffer list), executes *forms* in sequence, then restores the previously selected window and current buffer. It is just like **save-selected-window**, except that it explicitly selects *window*, also without altering the buffer list sequence.

The following functions choose one of the windows on the screen, offering various criteria for the choice.

**get-lru-window &optional frame dedicated** [Function]

This function returns the window least recently “used” (that is, selected). If any full-width windows are present, it only considers these. The selected window is always the most recently used window.

The selected window can be the least recently used window if it is the only window. A newly created window becomes the least recently used window until it is selected. A minibuffer window is never a candidate. Dedicated windows are never candidates unless the *dedicated* argument is non-*nil*, so if all existing windows are dedicated, the value is *nil*.

The argument *frame* controls which windows are considered.

- If it is `nil`, consider windows on the selected frame.
- If it is `t`, consider windows on all frames.
- If it is `visible`, consider windows on all visible frames.
- If it is `0`, consider windows on all visible or iconified frames.
- If it is a frame, consider windows on that frame.

**get-largest-window** &optional *frame dedicated*

[Function]

This function returns the window with the largest area (height times width). If there are no side-by-side windows, then this is the window with the most lines. A minibuffer window is never a candidate. Dedicated windows are never candidates unless the *dedicated* argument is non-`nil`, so if all existing windows are dedicated, the value is `nil`.

If there are two candidate windows of the same size, this function prefers the one that comes first in the cyclic ordering of windows (see following section), starting from the selected window.

The argument *frame* controls which set of windows to consider. See `get-lru-window`, above.

**get-window-with-predicate** *predicate* &optional *minibuf all-frames*

[Function]

*default*

This function returns a window satisfying *predicate*. It cycles through all visible windows using `walk-windows` (see Section 28.5 [Cyclic Window Ordering], page 503), calling *predicate* on each one of them with that window as its argument. The function returns the first window for which *predicate* returns a non-`nil` value; if that never happens, it returns *default*.

The optional arguments *minibuf* and *all-frames* specify the set of windows to include in the scan. See the description of `next-window` in Section 28.5 [Cyclic Window Ordering], page 503, for details.

## 28.5 Cyclic Ordering of Windows

When you use the command `C-x o` (`other-window`) to select the next window, it moves through all the windows on the screen in a specific cyclic order. For any given configuration of windows, this order never varies. It is called the *cyclic ordering of windows*.

This ordering generally goes from top to bottom, and from left to right. But it may go down first or go right first, depending on the order in which the windows were split.

If the first split was vertical (into windows one above each other), and then the sub-windows were split horizontally, then the ordering is left to right in the top of the frame, and then left to right in the next lower part of the frame, and so on. If the first split was horizontal, the ordering is top to bottom in the left part, and so on. In general, within each set of siblings at any level in the window tree, the order is left to right, or top to bottom.

**next-window** &optional *window minibuf all-frames*

[Function]

This function returns the window following *window* in the cyclic ordering of windows.

This is the window that `C-x o` would select if typed when *window* is selected. If

*window* is the only window visible, then this function returns *window*. If omitted, *window* defaults to the selected window.

The value of the argument *minibuf* determines whether the minibuffer is included in the window order. Normally, when *minibuf* is `nil`, the minibuffer is included if it is currently active; this is the behavior of `C-x o`. (The minibuffer window is active while the minibuffer is in use. See Chapter 20 [Minibuffers], page 278.)

If *minibuf* is `t`, then the cyclic ordering includes the minibuffer window even if it is not active.

If *minibuf* is neither `t` nor `nil`, then the minibuffer window is not included even if it is active.

The argument *all-frames* specifies which frames to consider. Here are the possible values and their meanings:

<code>nil</code>	Consider all the windows in <i>window</i> 's frame, plus the minibuffer used by that frame even if it lies in some other frame. If the minibuffer counts (as determined by <i>minibuf</i> ), then all windows on all frames that share that minibuffer count too.
<code>t</code>	Consider all windows in all existing frames.
<code>visible</code>	Consider all windows in all visible frames. (To get useful results, you must ensure <i>window</i> is in a visible frame.)
<code>0</code>	Consider all windows in all visible or iconified frames.
<code>a frame</code>	Consider all windows on that frame.
anything else	Consider precisely the windows in <i>window</i> 's frame, and no others.

This example assumes there are two windows, both displaying the buffer '`windows.texi`':

```
(selected-window)
  ⇒ #<window 56 on windows.texi>
(next-window (selected-window))
  ⇒ #<window 52 on windows.texi>
(next-window (next-window (selected-window)))
  ⇒ #<window 56 on windows.texi>
```

**previous-window** &**optional** *window minibuf all-frames* [Function]

This function returns the window preceding *window* in the cyclic ordering of windows. The other arguments specify which windows to include in the cycle, as in `next-window`.

**other-window** *count &optional all-frames* [Command]

This function selects the *count*th following window in the cyclic order. If *count* is negative, then it moves back  $-|count|$  windows in the cycle, rather than forward. It returns `nil`.

The argument *all-frames* has the same meaning as in `next-window`, but the *minibuf* argument of `next-window` is always effectively `nil`.

In an interactive call, *count* is the numeric prefix argument.

**walk-windows** *proc &optional minibuf all-frames* [Function]

This function cycles through all windows. It calls the function *proc* once for each window, with the window as its sole argument.

The optional arguments *minibuf* and *all-frames* specify the set of windows to include in the scan. See **next-window**, above, for details.

**window-list** *&optional frame minibuf window* [Function]

This function returns a list of the windows on *frame*, starting with *window*. If *frame* is **nil** or omitted, **window-list** uses the selected frame instead; if *window* is **nil** or omitted, it uses the selected window.

The value of *minibuf* determines if the minibuffer window is included in the result list. If *minibuf* is **t**, the result always includes the minibuffer window. If *minibuf* is **nil** or omitted, that includes the minibuffer window if it is active. If *minibuf* is neither **nil** nor **t**, the result never includes the minibuffer window.

## 28.6 Buffers and Windows

This section describes low-level functions to examine windows or to display buffers in windows in a precisely controlled fashion. See the following section for related functions that find a window to use and specify a buffer for it. The functions described there are easier to use than these, but they employ heuristics in choosing or creating a window; use these functions when you need complete control.

**set-window-buffer** *window buffer-or-name &optional keep-margins* [Function]

This function makes *window* display *buffer-or-name* as its contents. It returns **nil**. *buffer-or-name* must be a buffer, or the name of an existing buffer. This is the fundamental primitive for changing which buffer is displayed in a window, and all ways of doing that call this function.

```
(set-window-buffer (selected-window) "foo")
⇒ nil
```

Normally, displaying *buffer* in *window* resets the window's display margins, fringe widths, scroll bar settings, and position based on the local variables of *buffer*. However, if *keep-margins* is non-**nil**, the display margins and fringe widths of *window* remain unchanged. See Section 38.13 [Fringes], page 776.

**buffer-display-count** [Variable]

This buffer-local variable records the number of times a buffer is displayed in a window. It is incremented each time **set-window-buffer** is called for the buffer.

**window-buffer** *&optional window* [Function]

This function returns the buffer that *window* is displaying. If *window* is omitted, this function returns the buffer for the selected window.

```
(window-buffer)
⇒ #<buffer windows.texi>
```

**get-buffer-window** *buffer-or-name &optional all-frames* [Function]

This function returns a window currently displaying *buffer-or-name*, or **nil** if there is none. If there are several such windows, then the function returns the first one in

the cyclic ordering of windows, starting from the selected window. See Section 28.5 [Cyclic Window Ordering], page 503.

The argument *all-frames* controls which windows to consider.

- If it is `nil`, consider windows on the selected frame.
- If it is `t`, consider windows on all frames.
- If it is `visible`, consider windows on all visible frames.
- If it is `0`, consider windows on all visible or iconified frames.
- If it is a frame, consider windows on that frame.

**get-buffer-window-list** *buffer-or-name* &optional *minibuf* *all-frames* [Function]

This function returns a list of all the windows currently displaying *buffer-or-name*.

The two optional arguments work like the optional arguments of `next-window` (see Section 28.5 [Cyclic Window Ordering], page 503); they are *not* like the single optional argument of `get-buffer-window`. Perhaps we should change `get-buffer-window` in the future to make it compatible with the other functions.

**buffer-display-time** [Variable]

This variable records the time at which a buffer was last made visible in a window. It is always local in each buffer; each time `set-window-buffer` is called, it sets this variable to `(current-time)` in the specified buffer (see Section 39.5 [Time of Day], page 824). When a buffer is first created, `buffer-display-time` starts out with the value `nil`.

## 28.7 Displaying Buffers in Windows

In this section we describe convenient functions that choose a window automatically and use it to display a specified buffer. These functions can also split an existing window in certain circumstances. We also describe variables that parameterize the heuristics used for choosing a window. See the preceding section for low-level functions that give you more precise control. All of these functions work by calling `set-window-buffer`.

Do not use the functions in this section in order to make a buffer current so that a Lisp program can access or modify it; they are too drastic for that purpose, since they change the display of buffers in windows, which would be gratuitous and surprise the user. Instead, use `set-buffer` and `save-current-buffer` (see Section 27.2 [Current Buffer], page 481), which designate buffers as current for programmed access without affecting the display of buffers in windows.

**switch-to-buffer** *buffer-or-name* &optional *norecord* [Command]

This function makes *buffer-or-name* the current buffer, and also displays the buffer in the selected window. This means that a human can see the buffer and subsequent keyboard commands will apply to it. Contrast this with `set-buffer`, which makes *buffer-or-name* the current buffer but does not display it in the selected window. See Section 27.2 [Current Buffer], page 481.

If *buffer-or-name* does not identify an existing buffer, then a new buffer by that name is created. The major mode for the new buffer is set according to the variable `default-major-mode`. See Section 23.2.3 [Auto Major Mode], page 388. If *buffer-or-name* is `nil`, `switch-to-buffer` chooses a buffer using `other-buffer`.

Normally the specified buffer is put at the front of the buffer list (both the selected frame's buffer list and the frame-independent buffer list). This affects the operation of `other-buffer`. However, if `norecord` is non-`nil`, this is not done. See Section 27.8 [The Buffer List], page 490.

The `switch-to-buffer` function is often used interactively, as the binding of `C-x b`. It is also used frequently in programs. It returns the buffer that it switched to.

The next two functions are similar to `switch-to-buffer`, except for the described features.

**switch-to-buffer-other-window** *buffer-or-name* &optional *norecord* [Command]

This function makes *buffer-or-name* the current buffer and displays it in a window not currently selected. It then selects that window. The handling of the buffer is the same as in `switch-to-buffer`.

The currently selected window is absolutely never used to do the job. If it is the only window, then it is split to make a distinct window for this purpose. If the selected window is already displaying the buffer, then it continues to do so, but another window is nonetheless found to display it in as well.

This function updates the buffer list just like `switch-to-buffer` unless `norecord` is non-`nil`.

**pop-to-buffer** *buffer-or-name* &optional *other-window* *norecord* [Function]

This function makes *buffer-or-name* the current buffer and switches to it in some window, preferably not the window previously selected. The “popped-to” window becomes the selected window within its frame. The return value is the buffer that was switched to. If *buffer-or-name* is `nil`, that means to choose some other buffer, but you don't specify which.

If the variable `pop-up-frames` is non-`nil`, `pop-to-buffer` looks for a window in any visible frame already displaying the buffer; if there is one, it returns that window and makes it be selected within its frame. If there is none, it creates a new frame and displays the buffer in it.

If `pop-up-frames` is `nil`, then `pop-to-buffer` operates entirely within the selected frame. (If the selected frame has just a minibuffer, `pop-to-buffer` operates within the most recently selected frame that was not just a minibuffer.)

If the variable `pop-up-windows` is non-`nil`, windows may be split to create a new window that is different from the original window. For details, see Section 28.8 [Choosing Window], page 508.

If *other-window* is non-`nil`, `pop-to-buffer` finds or creates another window even if *buffer-or-name* is already visible in the selected window. Thus *buffer-or-name* could end up displayed in two windows. On the other hand, if *buffer-or-name* is already displayed in the selected window and *other-window* is `nil`, then the selected window is considered sufficient display for *buffer-or-name*, so that nothing needs to be done.

All the variables that affect `display-buffer` affect `pop-to-buffer` as well. See Section 28.8 [Choosing Window], page 508.

If *buffer-or-name* is a string that does not name an existing buffer, a buffer by that name is created. The major mode for the new buffer is set according to the variable `default-major-mode`. See Section 23.2.3 [Auto Major Mode], page 388.

This function updates the buffer list just like `switch-to-buffer` unless `norecord` is non-`nil`.

**replace-buffer-in-windows** *buffer-or-name*

[Command]

This function replaces *buffer-or-name* with some other buffer in all windows displaying it. It chooses the other buffer with `other-buffer`. In the usual applications of this function, you don't care which other buffer is used; you just want to make sure that *buffer-or-name* is no longer displayed.

This function returns `nil`.

## 28.8 Choosing a Window for Display

This section describes the basic facility that chooses a window to display a buffer in—`display-buffer`. All the higher-level functions and commands use this subroutine. Here we describe how to use `display-buffer` and how to customize it.

**display-buffer** *buffer-or-name* &**optional** *not-this-window frame*

[Command]

This command makes *buffer-or-name* appear in some window, like `pop-to-buffer`, but it does not select that window and does not make the buffer current. The identity of the selected window is unaltered by this function. *buffer-or-name* must be a buffer, or the name of an existing buffer.

If *not-this-window* is non-`nil`, it means to display the specified buffer in a window other than the selected one, even if it is already on display in the selected window. This can cause the buffer to appear in two windows at once. Otherwise, if *buffer-or-name* is already being displayed in any window, that is good enough, so this function does nothing.

`display-buffer` returns the window chosen to display *buffer-or-name*.

If the argument *frame* is non-`nil`, it specifies which frames to check when deciding whether the buffer is already displayed. If the buffer is already displayed in some window on one of these frames, `display-buffer` simply returns that window. Here are the possible values of *frame*:

- If it is `nil`, consider windows on the selected frame. (Actually, the last non-minibuffer frame.)
- If it is `t`, consider windows on all frames.
- If it is `visible`, consider windows on all visible frames.
- If it is `0`, consider windows on all visible or iconified frames.
- If it is a frame, consider windows on that frame.

Precisely how `display-buffer` finds or creates a window depends on the variables described below.

**display-buffer-reuse-frames**

[User Option]

If this variable is non-`nil`, `display-buffer` searches existing frames for a window displaying the buffer. If the buffer is already displayed in a window in some frame, `display-buffer` makes the frame visible and raises it, to use that window. If the buffer is not already displayed, or if `display-buffer-reuse-frames` is `nil`, `display-buffer`'s behavior is determined by other variables, described below.

**pop-up-windows**

[User Option]

This variable controls whether `display-buffer` makes new windows. If it is non-`nil` and there is only one window, then that window is split. If it is `nil`, then `display-buffer` does not split the single window, but uses it whole.

**split-height-threshold**

[User Option]

This variable determines when `display-buffer` may split a window, if there are multiple windows. `display-buffer` always splits the largest window if it has at least this many lines. If the largest window is not this tall, it is split only if it is the sole window and `pop-up-windows` is non-`nil`.

**even-window-heights**

[User Option]

This variable determines if `display-buffer` should even out window heights if the buffer gets displayed in an existing window, above or beneath another existing window. If `even-window-heights` is `t`, the default, window heights will be evened out. If `even-window-heights` is `nil`, the original window heights will be left alone.

**pop-up-frames**

[User Option]

This variable controls whether `display-buffer` makes new frames. If it is non-`nil`, `display-buffer` looks for an existing window already displaying the desired buffer, on any visible frame. If it finds one, it returns that window. Otherwise it makes a new frame. The variables `pop-up-windows` and `split-height-threshold` do not matter if `pop-up-frames` is non-`nil`.

If `pop-up-frames` is `nil`, then `display-buffer` either splits a window or reuses one.

See Chapter 29 [Frames], page 529, for more information.

**pop-up-frame-function**

[User Option]

This variable specifies how to make a new frame if `pop-up-frames` is non-`nil`.

Its value should be a function of no arguments. When `display-buffer` makes a new frame, it does so by calling that function, which should return a frame. The default value of the variable is a function that creates a frame using parameters from `pop-up-frame-alist`.

**pop-up-frame-alist**

[User Option]

This variable holds an alist specifying frame parameters used when `display-buffer` makes a new frame. See Section 29.3 [Frame Parameters], page 531, for more information about frame parameters.

**special-display-buffer-names**

[User Option]

A list of buffer names for buffers that should be displayed specially. If the buffer's name is in this list, `display-buffer` handles the buffer specially.

By default, special display means to give the buffer a dedicated frame.

If an element is a list, instead of a string, then the CAR of the list is the buffer name, and the rest of the list says how to create the frame. There are two possibilities for the rest of the list (its CDR). It can be an alist, specifying frame parameters, or it can contain a function and arguments to give to it. (The function's first argument is always the buffer to be displayed; the arguments from the list come after that.)

For example:

```
(("myfile" (minibuffer) (menu-bar-lines . 0)))
```

specifies to display a buffer named ‘myfile’ in a dedicated frame with specified **minibuffer** and **menu-bar-lines** parameters.

The list of frame parameters can also use the phony frame parameters **same-frame** and **same-window**. If the specified frame parameters include (**same-window . value**) and **value** is non-**nil**, that means to display the buffer in the current selected window. Otherwise, if they include (**same-frame . value**) and **value** is non-**nil**, that means to display the buffer in a new window in the currently selected frame.

**special-display-regexp**

[User Option]

A list of regular expressions that specify buffers that should be displayed specially. If the buffer’s name matches any of the regular expressions in this list, **display-buffer** handles the buffer specially.

By default, special display means to give the buffer a dedicated frame.

If an element is a list, instead of a string, then the CAR of the list is the regular expression, and the rest of the list says how to create the frame. See above, under **special-display-buffer-names**.

**special-display-p** *buffer-name*

[Function]

This function returns **non-nil** if displaying a buffer named *buffer-name* with **display-buffer** would create a special frame. The value is **t** if it would use the default frame parameters, or else the specified list of frame parameters.

**special-display-function**

[Variable]

This variable holds the function to call to display a buffer specially. It receives the buffer as an argument, and should return the window in which it is displayed.

The default value of this variable is **special-display-popup-frame**.

**special-display-popup-frame** *buffer &optional args*

[Function]

This function makes *buffer* visible in a frame of its own. If *buffer* is already displayed in a window in some frame, it makes the frame visible and raises it, to use that window. Otherwise, it creates a frame that will be dedicated to *buffer*. This function returns the window it used.

If *args* is an alist, it specifies frame parameters for the new frame.

If *args* is a list whose CAR is a symbol, then (**car args**) is called as a function to actually create and set up the frame; it is called with *buffer* as first argument, and (**cdr args**) as additional arguments.

This function always uses an existing window displaying *buffer*, whether or not it is in a frame of its own; but if you set up the above variables in your init file, before *buffer* was created, then presumably the window was previously made by this function.

**special-display-frame-alist**

[User Option]

This variable holds frame parameters for **special-display-popup-frame** to use when it creates a frame.

**same-window-buffer-names**

[User Option]

A list of buffer names for buffers that should be displayed in the selected window. If the buffer’s name is in this list, **display-buffer** handles the buffer by switching to it in the selected window.

**same-window-regexp**

[User Option]

A list of regular expressions that specify buffers that should be displayed in the selected window. If the buffer's name matches any of the regular expressions in this list, `display-buffer` handles the buffer by switching to it in the selected window.

**same-window-p *buffer-name***

[Function]

This function returns `t` if displaying a buffer named *buffer-name* with `display-buffer` would put it in the selected window.

**display-buffer-function**

[Variable]

This variable is the most flexible way to customize the behavior of `display-buffer`. If it is non-`nil`, it should be a function that `display-buffer` calls to do the work. The function should accept two arguments, the first two arguments that `display-buffer` received. It should choose or create a window, display the specified buffer in it, and then return the window.

This hook takes precedence over all the other options and hooks described above.

A window can be marked as “dedicated” to its buffer. Then `display-buffer` will not try to use that window to display any other buffer.

**window-dedicated-p *window***

[Function]

This function returns non-`nil` if *window* is marked as dedicated; otherwise `nil`.

**set-window-dedicated-p *window flag***

[Function]

This function marks *window* as dedicated if *flag* is non-`nil`, and nondedicated otherwise.

## 28.9 Windows and Point

Each window has its own value of point, 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.

See Chapter 30 [Positions], page 559, for more details on buffer positions.

As far as the user is concerned, point is where the cursor is, and when the user switches to another buffer, the cursor jumps to the position of point in that buffer.

**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. If *window* is `nil`, the selected window is used.

When *window* is the selected window and its buffer is also the current buffer, the value returned is the same as point in that 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, and its buffer is current, this simply does `goto-char`.

## 28.10 The Window Start Position

Each window contains 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.

**window-start** &optional window [Function]

This function returns the display-start position of window *window*. If *window* is `nil`, the selected window is used. For example,

```
(window-start)
⇒ 7058
```

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 1 if the buffer doesn't have any.

Redisplay updates the window-start position (if you have not specified it explicitly since the previous redisplay)—for example, 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.

For a realistic example of using `window-start`, see the description of `count-lines`. See [Definition of `count-lines`], page 563.

**window-end** &optional window update [Function]

This function returns the position of the end of the display in window *window*. If *window* is `nil`, the selected window is used.

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 the window ends, based on the current `window-start` value. If the saved value is valid, `window-end` returns that; 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.

**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 change the display-start position (that is, scroll the window) whenever 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 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 2, then point would 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)
 (1+ (window-start)))
⇒ 2
```

```
;; Here is what ‘foo’ looks like after executing
;;   the set-window-start expression.
----- Buffer: foo -----
his is the contents of buffer foo.
2
3
*4
5
6
----- Buffer: foo -----
```

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.

**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*, to the selected window.

If *position* is *t*, that means to check the last visible position in *window*.

The *pos-visible-in-window-p* 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 28.13 [Horizontal Scrolling], page 518.

If *position* is visible, *pos-visible-in-window-p* returns *t* if *partially* is *nil*; if *partially* is non-*nil*, and the character after *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 the *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))
```

**window-line-height** &optional *line* *window* [Function]

This function returns information about 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 argument *line* defaults to the current line in *window*; *window*, to 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.

## 28.11 Textual Scrolling

*Textual scrolling* means moving the text up or down through a window. It works by changing the value of the window’s `display-start` location. It may also change the value of `window-point` to keep point on the screen.

Textual scrolling was formerly called “vertical scrolling,” but we changed its name to distinguish it from the new vertical fractional scrolling feature (see Section 28.12 [Vertical Scrolling], page 518).

In the commands `scroll-up` and `scroll-down`, the directions “up” and “down” refer to the motion of the text in the buffer at which you are looking through 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 text in the middle of a buffer and repeatedly call `scroll-down`, you will eventually see the beginning of the buffer.

Some people have urged that the opposite convention be used: they imagine that the window moves over text that remains in place. Then “down” commands would take you to the end of the buffer. This view is more consistent with the actual relationship between windows and the text in the buffer, but it is less like what the user sees. The position of a window on the terminal does not move, and short scrolling commands clearly move the text up or down on the screen. We have chosen names that fit the user’s point of view.

The textual scrolling functions (aside from `scroll-other-window`) have unpredictable results if the current buffer is different from the buffer that is displayed in the selected window. See Section 27.2 [Current Buffer], page 481.

If the window contains a row which is taller than the height of the window (for example in the presence of a large image), the scroll functions will adjust the window `vscroll` to scroll the partially visible row. To disable this feature, Lisp code may bind the variable ‘`auto-window-vscroll`’ to `nil` (see Section 28.12 [Vertical Scrolling], page 518).

### `scroll-up` &optional `count`

[Command]

This function scrolls the text in the selected window upward `count` lines. If `count` is negative, scrolling is actually downward.

If `count` is `nil` (or omitted), then the length of scroll is `next-screen-context-lines` lines less than the usable height of the window (not counting its mode line).

`scroll-up` returns `nil`, unless it gets an error because it can’t scroll any further.

### `scroll-down` &optional `count`

[Command]

This function scrolls the text in the selected window downward `count` lines. If `count` is negative, scrolling is actually upward.

If `count` is omitted or `nil`, then the length of the scroll is `next-screen-context-lines` lines less than the usable height of the window (not counting its mode line).

`scroll-down` returns `nil`, unless it gets an error because it can’t scroll any further.

**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 one at the top left corner. 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 302.

When the minibuffer is active, it is the next window if the selected window is the one at the bottom right corner. In this case, `scroll-other-window` attempts to scroll the minibuffer. If the minibuffer contains just one line, it has nowhere to scroll to, so the line reappears after the echo area momentarily displays the message ‘Beginning of buffer’.

**other-window-scroll-buffer** [Variable]

If this variable is non-`nil`, it tells `scroll-other-window` which buffer 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.

**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 action 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-up-aggressively**, a larger value scrolls more aggressively.

**scroll-step**

[User Option]

This variable is an older variant of **scroll-conservatively**. The difference is that it 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**, scrolling which would move the current point position out of the window chooses the new position of point so that the vertical position of the cursor is unchanged, if possible.

If it is **non-nil** and not **t**, then the scrolling functions always preserve the vertical position of point, if possible.

**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.

**recenter &optional count**

[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 nonnegative 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 a **non-nil** list, then it stands for the line in the middle of the window.

If *count* is **nil**, **recenter** puts the line containing point in the middle of the window, then clears and redisplay the entire selected frame.

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. This action is so handy that some people make a separate key binding to do this. For example,

```
(defun line-to-top-of-window ()
  "Scroll current line to top of window.
Replaces three keystroke sequence C-u 0 C-l."
  (interactive)
  (recenter 0))

(global-set-key [kp-multiply] 'line-to-top-of-window)
```

## 28.12 Vertical Fractional Scrolling

*Vertical fractional scrolling* means shifting the image in the window up or down by a specified multiple or fraction of a line. 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. 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 are 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*. If *window* is `nil`, the selected window is used. 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

[Function]

This function sets *window*'s vertical scroll position to *lines*. The argument *lines* should be zero or positive; if not, it is taken as zero.

If *window* is `nil`, the selected window is used.

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*.

### auto-window-vscroll

[Variable]

If this variable is non-`nil`, the line-move, scroll-up, and scroll-down functions will automatically modify the window vscroll to scroll through display rows that are taller than the height of the window, for example in the presence of large images.

## 28.13 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.

**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).

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.

The value is never negative. It is zero when no horizontal scrolling has been done in *window* (which is usually the case).

If *window* is `nil`, the selected window is used.

```
(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. 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)))))
```

## 28.14 The Size of a Window

An Emacs window is rectangular, and its size information consists of the height (the number of lines) and the width (the number of character positions in each line). The mode line is included in the height. But the width does not count the scroll bar or the column of ‘|’ characters that separates side-by-side windows.

The following three functions return size information about a window:

**window-height** &optional *window* [Function]

This function returns the number of lines in *window*, including its mode line and header line, if any. If *window* fills its entire frame except for the echo area, this is typically one less than the value of **frame-height** on that frame.

If *window* is **nil**, the function uses the selected window.

```
(window-height)
  ⇒ 23
```

```
(split-window-vertically)
  ⇒ #<window 4 on windows.texi>
(window-height)
  ⇒ 11
```

**window-body-height &optional window** [Function]

Like `window-height` but the value does not include the mode line (if any) or the header line (if any).

**window-width &optional window** [Function]

This function returns the number of columns in `window`. If `window` fills its entire frame, this is the same as the value of `frame-width` on that frame. The width does not include the window's scroll bar or the column of ‘|’ characters that separates side-by-side windows.

If `window` is `nil`, the function uses the selected window.

```
(window-width)
  ⇒ 80
```

**window-edges &optional window** [Function]

This function returns a list of the edge coordinates of `window`. If `window` is `nil`, the selected window is used.

The order of the list is (*left top right bottom*), all elements relative to 0, 0 at the top left corner of the frame. The element *right* of the value is one more than the rightmost column used by `window`, and *bottom* is one more than the bottommost row used by `window` and its mode-line.

The edges include the space used by the window's scroll bar, display margins, fringes, header line, and mode line, if it has them. Also, if the window has a neighbor on the right, its right edge value includes the width of the separator line between the window and that neighbor. Since the width of the window does not include this separator, the width does not usually equal the difference between the right and left edges.

**window-inside-edges &optional window** [Function]

This is similar to `window-edges`, but the edge values it returns include only the text area of the window. They do not include the header line, mode line, scroll bar or vertical separator, fringes, or display margins.

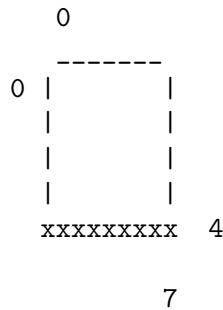
Here are the results obtained on a typical 24-line terminal with just one window, with menu bar enabled:

```
(window-edges (selected-window))
  ⇒ (0 1 80 23)
(window-inside-edges (selected-window))
  ⇒ (0 1 80 22)
```

The bottom edge is at line 23 because the last line is the echo area. The bottom inside edge is at line 22, which is the window's mode line.

If `window` is at the upper left corner of its frame, and there is no menu bar, then *bottom* returned by `window-edges` is the same as the value of `(window-height)`, *right* is almost the same as the value of `(window-width)`, and *top* and *left* are zero. For example, the edges

of the following window are ‘0 0 8 5’. Assuming that the frame has more than 8 columns, the last column of the window (column 7) holds a border rather than text. The last row (row 4) holds the mode line, shown here with ‘xxxxxxxxx’.



In the following example, let's suppose that the frame is 7 columns wide. Then the edges of the left window are '0 0 4 3' and the edges of the right window are '4 0 7 3'. The inside edges of the left window are '0 0 3 2', and the inside edges of the right window are '4 0 7 2',



0 34 7

**window-pixel-edges** &optional window

### [Function]

This function is like `window.edges` except that, on a graphical display, the edge values are measured in pixels instead of in character lines and columns.

`window-inside-pixel-edges` &optional `window`

### [Function]

This function is like `window-inside-edges` except that, on a graphical display, the edge values are measured in pixels instead of in character lines and columns.

## 28.15 Changing the Size of a Window

The window size functions fall into two classes: high-level commands that change the size of windows and low-level functions that access window size. Emacs does not permit overlapping windows or gaps between windows, so resizing one window affects other windows.

`enlarge-window-size` & optional `horizontal`

[Command]

This function makes the selected window size lines taller, stealing lines from neighboring windows. It takes the lines from one window at a time until that window is used up, then takes from another. If a window from which lines are stolen shrinks below `window-min-height` lines, that window disappears.

If `horizontal` is non-`nil`, this function makes `window` wider by `size` columns, stealing columns instead of lines. If a window from which columns are stolen shrinks below `window-min-width` columns, that window disappears.

If the requested size would exceed that of the window's frame, then the function makes the window occupy the entire height (or width) of the frame.

If there are various other windows from which lines or columns can be stolen, and some of them specify fixed size (using `window-size-fixed`, see below), they are left untouched while other windows are “robbed.” If it would be necessary to alter the size of a fixed-size window, `enlarge-window` gets an error instead.

If `size` is negative, this function shrinks the window by `-size` lines or columns. If that makes the window smaller than the minimum size (`window-min-height` and `window-min-width`), `enlarge-window` deletes the window.

`enlarge-window` returns `nil`.

**enlarge-window-horizontally** *columns* [Command]

This function makes the selected window *columns* wider. It could be defined as follows:

```
(defun enlarge-window-horizontally (columns)
  (interactive "p")
  (enlarge-window columns t))
```

**shrink-window** *size* &optional *horizontal* [Command]

This function is like `enlarge-window` but negates the argument `size`, making the selected window smaller by giving lines (or columns) to the other windows. If the window shrinks below `window-min-height` or `window-min-width`, then it disappears.

If `size` is negative, the window is enlarged by `-size` lines or columns.

**shrink-window-horizontally** *columns* [Command]

This function makes the selected window *columns* narrower. It could be defined as follows:

```
(defun shrink-window-horizontally (columns)
  (interactive "p")
  (shrink-window columns t))
```

**adjust-window-trailing-edge** *window delta horizontal* [Function]

This function makes the selected window *delta* lines taller or *delta* columns wider, by moving the bottom or right edge. This function does not delete other windows; if it cannot make the requested size adjustment, it signals an error. On success, this function returns `nil`.

**fit-window-to-buffer** &optional *window max-height min-height* [Function]

This function makes *window* the right height to display its contents exactly. If *window* is omitted or `nil`, it uses the selected window.

The argument `max-height` specifies the maximum height the window is allowed to be; `nil` means use the frame height. The argument `min-height` specifies the minimum height for the window; `nil` means use `window-min-height`. All these height values include the mode-line and/or header-line.

**shrink-window-if-larger-than-buffer** &optional *window* [Command]

This command shrinks *window* vertically to be as small as possible while still showing the full contents of its buffer—but not less than `window-min-height` lines. If *window* is not given, it defaults to the selected window.

However, the command does nothing if the window is already too small to display the whole text of the buffer, or if part of the contents are currently scrolled off screen, or if the window is not the full width of its frame, or if the window is the only window in its frame.

This command returns `non-nil` if it actually shrank the window and `nil` otherwise.

**window-size-fixed**

[Variable]

If this variable is `non-nil`, in any given buffer, then the size of any window displaying the buffer remains fixed unless you explicitly change it or Emacs has no other 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.

This variable automatically becomes buffer-local when set.

Explicit size-change functions such as `enlarge-window` get an error if they would have to change a window size which is fixed. Therefore, when you want to change the size of such a window, you should bind `window-size-fixed` to `nil`, like this:

```
(let ((window-size-fixed nil))
  (enlarge-window 10))
```

Note that changing the frame size will change the size of a fixed-size window, if there is no other alternative.

The following two variables constrain the window-structure-changing functions to a minimum height and width.

**window-min-height**

[User Option]

The value of this variable determines how short a window may become before it is automatically deleted. Making a window smaller than `window-min-height` automatically deletes it, and no window may be created shorter than this. The default value is 4.

The absolute minimum window height is one; actions that change window sizes reset this variable to one if it is less than one.

**window-min-width**

[User Option]

The value of this variable determines how narrow a window may become before it is automatically deleted. Making a window smaller than `window-min-width` automatically deletes it, and no window may be created narrower than this. The default value is 10.

The absolute minimum window width is two; actions that change window sizes reset this variable to two if it is less than two.

## 28.16 Coordinates and Windows

This section describes how to relate screen coordinates to windows.

**window-at x y &optional frame**

[Function]

This function returns the window containing the specified cursor position in the frame `frame`. The coordinates `x` and `y` are measured in characters and count from the top left corner of the frame. If they are out of range, `window-at` returns `nil`.

If you omit *frame*, the selected frame is used.

**coordinates-in-window-p** *coordinates* *window* [Function]

This function checks whether a particular frame position falls within the window *window*.

The argument *coordinates* is a cons cell of the form *(x . y)*. The coordinates *x* and *y* are measured in characters, and count from the top left corner of the screen or frame.

The value returned by **coordinates-in-window-p** is non-nil if the coordinates are inside *window*. The value also indicates what part of the window the position is in, as follows:

**(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*.

**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.

## 28.17 The Window Tree

A *window tree* specifies the layout, size, and relationship between all windows in one frame.

**window-tree** &optional *frame* [Function]

This function returns the window tree for frame *frame*. If *frame* is omitted, the selected frame is used.

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 not split, *root* is the root window itself. Otherwise, *root* is a list (*dir edges w1 w2 ...*) where *dir* is *nil* for a horizontal split, and *t* for a vertical split, *edges* gives the combined size and position of the subwindows in the split, and the rest of the elements are the subwindows in the split. Each of the subwindows may again be a window or a list representing a window split, and so on. The *edges* element is a list (*left top right bottom*) similar to the value returned by `window-edges`.

## 28.18 Window Configurations

A *window configuration* records the entire layout of one frame—all windows, their sizes, which buffers they contain, what part of each buffer is displayed, and the values of point and the mark; also their fringes, margins, and scroll bar settings. It also includes the values of `window-min-height`, `window-min-width` and `minibuffer-scroll-window`. An exception is made for point in the selected window for the current buffer; its value is not saved in the window configuration.

You can bring back an entire previous layout by restoring a window configuration previously saved. If you want to record all frames instead of just one, use a frame configuration instead of a window configuration. See Section 29.12 [Frame Configurations], page 546.

**current-window-configuration** &**optional** *frame* [Function]

This function returns a new object representing *frame*'s current window configuration. If *frame* is omitted, the selected frame is used.

**set-window-configuration** *configuration* [Function]

This function restores the configuration of windows and buffers as specified by *configuration*, for the frame that *configuration* was created for.

The argument *configuration* must be a value that was previously returned by `current-window-configuration`. This configuration is restored in the frame from which *configuration* was made, whether that frame is selected or not. This always counts as a window size change and triggers execution of the `window-size-change-functions` (see Section 28.19 [Window Hooks], page 527), because `set-window-configuration` doesn't know how to tell whether the new configuration actually differs from the old one.

If the frame which *configuration* was saved from is dead, all this function does is restore the three variables `window-min-height`, `window-min-width` and `minibuffer-scroll-window`. In this case, the function returns *nil*. Otherwise, it returns *t*.

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-vertically nil)
           ...)
    (set-window-configuration config)))
```

**save-window-excursion** *forms...* [Special Form]

This special form records the window configuration, executes *forms* in sequence, then restores the earlier window configuration. The window configuration includes, for each window, the value of point and the portion of the buffer that is visible. It also

includes the choice of selected window. However, it does not include the value of point in the current buffer; use `save-excursion` also, if you wish to preserve that. Don't use this construct when `save-selected-window` is sufficient.

Exit from `save-window-excursion` always triggers execution of the `window-size-change-functions`. (It doesn't know how to tell whether the restored configuration actually differs from the one in effect at the end of the *forms*.)

The return value is the value of the final form in *forms*. For example:

```
(split-window)
  ⇒ #<window 25 on control.texi>
(setq w (selected-window))
  ⇒ #<window 19 on control.texi>
(save-window-excursion
  (delete-other-windows w)
  (switch-to-buffer "foo")
  'do-something)
  ⇒ do-something
;; The screen is now split again.
```

`window-configuration-p object` [Function]

This function returns `t` if *object* is a window configuration.

`compare-window-configurations config1 config2` [Function]

This function compares two window configurations as regards the structure of windows, but ignores the values of point and mark and the saved scrolling positions—it can return `t` even if those aspects differ.

The function `equal` can also compare two window configurations; it regards configurations as unequal if they differ in any respect, even a saved point or mark.

`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.

## 28.19 Hooks for Window Scrolling and Changes

This section describes how a Lisp program can take action whenever a window displays a different part of its buffer or a different buffer. There are three actions that can change this: scrolling the window, switching buffers in the window, and changing the size of the window. The first two actions run `window-scroll-functions`; the last runs `window-size-change-functions`.

`window-scroll-functions` [Variable]

This variable holds a list of functions that Emacs should call before redisplaying a window with scrolling. It is not a normal hook, because each function is called with two arguments: the window, and its new display-start position.

Displaying a different buffer in the window also runs these functions.

These functions must be careful in using `window-end` (see Section 28.10 [Window Start], page 512); 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.

**window-size-change-functions**

[Variable]

This variable holds a list of functions to be called if the size of any window changes for any reason. The functions are called just once per redisplay, and just once for each frame on which size changes have occurred.

Each function receives the frame as its sole argument. There is no direct way to find out which windows on that frame have changed size, or precisely how. However, if a size-change function records, at each call, the existing windows and their sizes, it can also compare the present sizes and the previous sizes.

Creating or deleting windows counts as a size change, and therefore causes these functions to be called. Changing the frame size also counts, because it changes the sizes of the existing windows.

It is not a good idea to use `save-window-excursion` (see Section 28.18 [Window Configurations], page 526) in these functions, because that always counts as a size change, and it would cause these functions to be called over and over. In most cases, `save-selected-window` (see Section 28.4 [Selecting Windows], page 502) is what you need here.

**redisplay-end-trigger-functions**

[Variable]

This abnormal hook is run whenever redisplay in a window uses text that extends past a specified end trigger position. You set the end trigger position with the function `set-window-redisplay-end-trigger`. The functions are called with two arguments: the window, and the end trigger position. Storing `nil` for the end trigger position turns off the feature, and the trigger value is automatically reset to `nil` just after the hook is run.

**set-window-redisplay-end-trigger** *window position*

[Function]

This function sets *window*'s end trigger position at *position*.

**window-redisplay-end-trigger** &**optional** *window*

[Function]

This function returns *window*'s current end trigger position. If *window* is `nil` or omitted, it uses the selected window.

**window-configuration-change-hook**

[Variable]

A normal hook that is run every time you change the window configuration of an existing frame. This includes splitting or deleting windows, changing the sizes of windows, or displaying a different buffer in a window. The frame whose window configuration has changed is the selected frame when this hook runs.

## 29 Frames

In Emacs editing, A *frame* is a screen object that contains one or more Emacs windows. It's the kind of object that is 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.

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. In Emacs Lisp, a *frame object* is a Lisp object that represents a frame on the screen.

When Emacs runs on a text-only terminal, it starts with one *terminal frame*. If you create additional ones, Emacs displays one and only one at any given time—on the terminal screen, of course.

When Emacs communicates directly with a supported window system, such as X, it does not have a terminal frame; instead, it starts with a single *window frame*, but you can create more, and Emacs can display several such frames at once as is usual for window systems.

**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:

- `x` The frame is displayed in an X window.
- `t` A terminal frame on a character display.
- `mac` The frame is displayed on a Macintosh.
- `w32` The frame is displayed on MS-Windows 9X/NT.
- `pc` The frame is displayed on an MS-DOS terminal.

See Chapter 38 [Display], page 739, for information about the related topic of controlling Emacs redisplay.

### 29.1 Creating Frames

To create a new frame, call the function `make-frame`.

**make-frame &optional alist** [Function]

This function creates and returns a new frame, displaying the current buffer. If you are using a supported window system, it makes a window frame; otherwise, it makes a terminal frame.

The argument is an alist specifying frame parameters. Any parameters not mentioned in *alist* default according to the value of the variable `default-frame-alist`; parameters not specified even there default from the standard X resources or whatever is used instead on your system.

The set of possible parameters depends in principle on what kind of window system Emacs uses to display its frames. See Section 29.3.3 [Window Frame Parameters], page 533, for documentation of individual parameters you can specify.

This function itself does not make the new frame the selected frame. See Section 29.9 [Input Focus], page 543. The previously selected frame remains selected. However, the window system may select the new frame for its own reasons, for instance if the frame appears under the mouse pointer and your setup is for focus to follow the pointer.

**before-make-frame-hook**

[Variable]

A normal hook run by `make-frame` before it actually creates the frame.

**after-make-frame-functions**

[Variable]

An abnormal hook run by `make-frame` after it creates the frame. Each function in `after-make-frame-functions` receives one argument, the frame just created.

## 29.2 Multiple Displays

A single Emacs can talk to more than one X display. Initially, Emacs uses just one display—the one chosen with the `DISPLAY` environment variable or with the ‘`--display`’ option (see section “Initial Options” in *The GNU Emacs Manual*). To connect to another display, use the command `make-frame-on-display` or specify the `display` frame parameter when you create the frame.

Emacs treats each X server as a separate terminal, giving each one 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 29.9 [Input Focus], page 543.

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 11.10 [Buffer-Local Variables], page 147) or frame-local.

A single X server can handle more than one screen. A display name ‘`host:server.screen`’ has three parts; the last part specifies the screen number for a given server. When you use two screens belonging to one server, Emacs knows by the similarity in their names that they share a single keyboard, and it treats them as a single terminal.

Note that some graphical terminals can output to more than one monitor (or other output device) at the same time. On these “multi-monitor” setups, a single `display` value controls the output to all the physical monitors. In this situation, there is currently no platform-independent way for Emacs to distinguish between the different physical monitors.

**make-frame-on-display display &optional parameters**

[Command]

This creates and returns a new frame on display `display`, taking the other frame parameters from `parameters`. Aside from the `display` argument, it is like `make-frame` (see Section 29.1 [Creating Frames], page 529).

**x-display-list**

[Function]

This 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`. It does not create a frame on that display, but it permits you to check that communication can be established with that 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. The values you specify override

the resource values recorded in the X server itself; they apply to all Emacs frames created on this display. Here's an example of what this string might look like:

```
"*BorderWidth: 3\n*InternalBorder: 2\n"
```

See section "X Resources" in *The GNU Emacs Manual*.

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 29.5 [Deleting Frames], page 541).

## 29.3 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 window systems. A terminal frame has a few parameters, mostly for compatibility's sake; 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.

### 29.3.1 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 parameters of 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.

**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.

### 29.3.2 Initial Frame Parameters

You can specify the parameters for the initial startup frame by setting `initial-frame-alist` in your init file (see Section 39.1.2 [Init File], page 813).

#### `initial-frame-alist`

[Variable]

This variable's value is an alist of parameter values used when creating the initial window 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 specify a separate minibuffer-only frame with `(minibuffer . nil)`, and you have not created one, Emacs creates one for you.

#### `minibuffer-frame-alist`

[Variable]

This variable's value is an alist of parameter values used when creating an initial minibuffer-only frame—if such a frame is needed, according to the parameters for the main initial frame.

#### `default-frame-alist`

[Variable]

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.

See also `special-display-frame-alist`. See [Definition of special-display-frame-alist], page 510.

If you use options that specify window appearance when you invoke Emacs, they take effect by adding elements to `default-frame-alist`. One exception is ‘`-geometry`’, which adds the specified position to `initial-frame-alist` instead. See section “Command Line Arguments for Emacs Invocation” in *The GNU Emacs Manual*.

### 29.3.3 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* in terminal frames.

#### 29.3.3.1 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:dpyscreen`", just like the `DISPLAY` environment variable.

**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, it appears in the window system's border for the frame, and also in the mode line of windows in that frame if `mode-line-frame-identification` uses '`%F`' (see Section 23.4.5 [%-Constructs], page 407). This is normally the case when Emacs is not using a window system, and can only display one frame at a time. See Section 29.4 [Frame Titles], page 540.

**name** The name of the frame. The frame name serves as a default for the frame title, if the `title` parameter is unspecified or `nil`. If you don't specify a name, Emacs sets the frame name automatically (see Section 29.4 [Frame Titles], page 540).

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.

#### 29.3.3.2 Position Parameters

Position parameters' values are normally measured in pixels, but on text-only terminals they count characters or lines instead.

**left** The screen position of the left edge, in pixels, with respect to the left edge of the screen. The value may be a positive number `pos`, or a list of the form `(+ pos)` which permits specifying a negative `pos` value.

A negative number `-pos`, or a list of the form `(- pos)`, actually specifies the position of the right edge of the window with respect to the right edge of the screen. A positive value of `pos` counts toward the left. **Reminder:** if the parameter is a negative integer `-pos`, then `pos` is positive.

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 well.

**top** The screen position of the top edge, in pixels, with respect to the top edge of the screen. 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 if and when the frame is iconified. If you specify a value for this parameter, then you must also specify a value for **icon-top** and vice versa. The window manager may ignore these two parameters.

**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 if and when the frame is iconified.

**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**.

### 29.3.3.3 Size Parameters

Size parameters' values are normally measured in pixels, but on text-only terminals they count characters or lines instead.

**height** The height of the frame contents, in characters. (To get the height in pixels, call **frame-pixel-height**; see Section 29.3.4 [Size and Position], page 538.)

**width** The width of the frame contents, in characters. (To get the height in pixels, call **frame-pixel-width**; see Section 29.3.4 [Size and Position], page 538.)

**user-size**

This does for the size parameters **height** and **width** what the **user-position** parameter (see above) does for the position parameters **top** and **left**.

**fullscreen**

Specify that width, height or both shall be set to the size of the screen. The value **fullwidth** specifies that width shall be the size of the screen. The value **fullheight** specifies that height shall be the size of the screen. The value **fullboth** specifies that both the width and the height shall be set to the size of the screen.

### 29.3.3.4 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 border.

**internal-border-width**

The distance in pixels between text (or fringe) and the frame's border.

**vertical-scroll-bars**

Whether the frame has scroll bars 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.

**scroll-bar-width**

The width of vertical scroll bars, in pixels, or `nil` meaning to use the default width.

**left-fringe****right-fringe**

The default width of the left and right fringes of windows in this frame (see Section 38.13 [Fringes], page 776). If either of these is zero, that effectively removes the corresponding fringe. A value of `nil` stands for the standard fringe width, which is the width needed to display the fringe bitmaps.

The combined fringe widths must add up to an integral number of columns, so the actual default fringe widths for the frame may be larger than the specified values. The extra width needed to reach an acceptable total is distributed evenly between the left and right fringe. However, you can force one fringe or the other to a precise width by specifying that width as a negative integer. If both widths are negative, only the left fringe gets the specified width.

**menu-bar-lines**

The number of lines to allocate at the top of the frame for a menu bar. The default is 1. A value of `nil` means don't display a menu bar. See Section 22.17.5 [Menu Bar], page 377. (The X toolkit and GTK allow at most one menu bar line; they treat larger values as 1.)

**tool-bar-lines**

The number of lines to use for the tool bar. A value of `nil` means don't display a tool bar. (GTK allows at most one tool bar line; it treats larger values as 1.)

**line-spacing**

Additional space to leave below each text line, in pixels (a positive integer). See Section 38.11 [Line Height], page 761, for more information.

### 29.3.3.5 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 new frame uses that minibuffer.

**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.

**29.3.3.6 Window Management Parameters**

These frame parameters, meaningful only on window system displays, interact with the window manager.

**visibility**

The state of visibility of the frame. There are three possibilities: **nil** for invisible, **t** for visible, and **icon** for iconified. See Section 29.10 [Visibility of Frames], page 545.

**auto-raise**

Whether selecting the frame raises it (non-**nil** means yes).

**auto-lower**

Whether deselecting the frame lowers it (non-**nil** means yes).

**icon-type**

The type of icon to use for this frame when it is iconified. If the value is a string, that specifies a file containing a bitmap to use. Any other non-**nil** value specifies the default bitmap icon (a picture of a gnu); **nil** specifies a text 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 number of the window-system window used by the frame to contain the actual Emacs windows.

**outer-window-id**

The number of the outermost window-system window used for the whole frame.

**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.

**29.3.3.7 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.)

<b>hollow</b>	Display a hollow box.
<b>nil</b>	Don't display a cursor.
<b>bar</b>	Display a vertical bar between characters.
<b>(bar . width)</b>	Display a vertical bar <i>width</i> pixels wide between characters.
<b>hbar</b>	Display a horizontal bar.
<b>(hbar . height)</b>	Display a horizontal bar <i>height</i> pixels high.

The buffer-local variable **cursor-type** overrides the value of the **cursor-type** frame parameter, but if it is **t**, that means to use the cursor specified for the frame.

#### **blink-cursor-alist** [Variable]

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, because the variable is examined only when you specify the **cursor-type** parameter.

### 29.3.3.8 Color Parameters

These frame parameters control the use of colors.

#### **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 in terminal frames. 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, a floating point number.

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.

These 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 38.12.9 [Fontsets], page 774). It is equivalent to the `font` attribute of the `default` face.

**foreground-color**

The color to use for the image of a character. 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.

### 29.3.4 Frame Size And Position

You can read or change the size and position of a frame using the frame parameters `left`, `top`, `height`, and `width`. Whatever geometry parameters you don’t specify are chosen by the window manager in its usual fashion.

Here are some special features for working with sizes and positions. (For the precise meaning of “selected frame” used by these functions, see Section 29.9 [Input Focus], page 543.)

**set-frame-position** *frame left top* [Function]

This function sets the position of the top left corner of *frame* to *left* and *top*. These arguments are measured in pixels, and normally count from the top left corner of the screen.

Negative parameter values position the bottom edge of the window up from the bottom edge of the screen, or the right window edge to the left of the right edge of the screen. It would probably be better if the values were always counted from the left and top, so that negative arguments would position the frame partly off the top or left edge of the screen, but it seems inadvisable to change that now.

**frame-height** &optional *frame* [Function]

**frame-width** &optional *frame* [Function]

These functions return the height and width of *frame*, measured in lines and columns.

If you don't supply *frame*, they use the selected frame.

**screen-height** [Function]

**screen-width** [Function]

These functions are old aliases for **frame-height** and **frame-width**. When you are using a non-window terminal, the size of the frame is normally the same as the size of the terminal screen.

**frame-pixel-height** &optional *frame* [Function]

**frame-pixel-width** &optional *frame* [Function]

These functions return the height and width of *frame*, measured in pixels. If you don't supply *frame*, they use the selected frame.

**frame-char-height** &optional *frame* [Function]

**frame-char-width** &optional *frame* [Function]

These functions return the height and width of a character in *frame*, measured in pixels. The values depend on the choice of font. If you don't supply *frame*, these functions use the selected frame.

**set-frame-size** *frame cols rows* [Function]

This function sets the size of *frame*, measured in characters; *cols* and *rows* specify the new width and height.

To set the size based on values measured in pixels, use **frame-char-height** and **frame-char-width** to convert them to units of characters.

**set-frame-height** *frame lines* &optional *pretend* [Function]

This function resizes *frame* to a height of *lines* lines. The sizes of existing windows in *frame* are altered proportionally to fit.

If *pretend* is non-nil, then Emacs displays *lines* lines of output in *frame*, but does not change its value for the actual height of the frame. This is only useful for a terminal frame. 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 "for real" does not always work, because knowing the correct actual size may be necessary for correct cursor positioning on a terminal frame.

**set-frame-width** *frame width &optional pretend* [Function]

This function sets the width of *frame*, measured in characters. The argument *pretend* has the same meaning as in **set-frame-height**.

The older functions **set-screen-height** and **set-screen-width** were used to specify the height and width of the screen, in Emacs versions that did not support multiple frames. They are semi-obsolete, but still work; they apply to the selected frame.

### 29.3.5 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. These are the *value* possibilities for the position parameters:

**an integer** A positive integer relates the left edge or top edge of the window to the left or top edge of the screen. A negative integer relates the right or bottom edge of the window to the right or bottom edge of the screen.

**(+ position)**

This specifies the position of the left or top edge of the window relative to the left or top edge of the screen. The integer *position* may be positive or negative; a negative value specifies a position outside the screen.

**(- position)**

This specifies the position of the right or bottom edge of the window relative to the right or bottom edge of the screen. The integer *position* may be positive or negative; a negative value specifies a position outside the screen.

Here is an example:

```
(x-parse-geometry "35x70+0-0")
⇒ ((height . 70) (width . 35)
    (top - 0) (left . 0))
```

## 29.4 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. The variable's value is actually a mode line construct, just like `mode-line-format`, except that the '%c' and '%l' constructs are ignored. See Section 23.4.2 [Mode Line Data], page 402.

**icon-title-format** [Variable]

This variable specifies how to compute the name for an iconified frame, when you have not explicitly specified the frame title. This title appears in the icon itself.

**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`.

## 29.5 Deleting Frames

Frames remain potentially visible until you explicitly *delete* them. A deleted frame cannot appear on the screen, but continues to exist as a Lisp object until there are no references to it.

**delete-frame &optional frame force** [Command]

This function deletes the frame `frame`. Unless `frame` is a tooltip, it first runs the hook `delete-frame-functions` (each function gets one argument, `frame`). By default, `frame` is the selected frame.

A frame cannot be deleted if its minibuffer is used by other frames. Normally, you cannot delete a frame if all other frames are invisible, but if the `force` is non-`nil`, then you are allowed to do so.

**frame-live-p frame** [Function]

The function `frame-live-p` returns non-`nil` if the frame `frame` has not been deleted. The possible non-`nil` return values are like those of `framep`. See Chapter 29 [Frames], page 529.

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 21.6.10 [Misc Events], page 322.

## 29.6 Finding All Frames

**frame-list** [Function]

The function `frame-list` returns a list of all the frames 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 29.10 [Visibility of Frames], page 545. (Terminal frames always count as “visible,” even though only the selected one is actually displayed.)

**next-frame &optional frame minibuf** [Function]

The function `next-frame` lets you cycle conveniently through all the frames on the current display from an arbitrary starting point. It returns the “next” frame after `frame` in the cycle. If `frame` is omitted or `nil`, it defaults to the selected frame (see Section 29.9 [Input Focus], page 543).

The second argument, `minibuf`, says which frames to consider:

- `nil` Exclude minibuffer-only frames.
- `visible` Consider all visible frames.
- `0` Consider all visible or iconified frames.
- `a window` Consider only the frames using that particular window as their minibuffer.
- 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 28.5 [Cyclic Window Ordering], page 503.

## 29.7 Frames and Windows

Each window is part of one and only one frame; you can get the frame with `window-frame`.

**window-frame window** [Function]

This function returns the frame that `window` is on.

All the non-minibuffer windows in a frame are arranged in a cyclic order. The order runs from the frame’s top window, which is at the upper left corner, down and to the right, until it reaches the window at the lower right corner (always the minibuffer window, if the frame has one), and then it moves back to the top. See Section 28.5 [Cyclic Window Ordering], page 503.

**frame-first-window &optional frame** [Function]

This returns the topmost, leftmost window of frame `frame`. If omitted or `nil`, `frame` defaults to the selected frame.

At any time, exactly one window on any frame is *selected within the frame*. The significance of this designation is that selecting the frame also selects this window. You can get the frame’s current selected window with `frame-selected-window`.

**frame-selected-window &optional frame** [Function]

This function returns the window on `frame` that is selected within `frame`. If omitted or `nil`, `frame` defaults to the selected frame.

**set-frame-selected-window** *frame window* [Function]

This sets the selected window of frame *frame* to *window*. If *frame* is *nil*, it operates on the selected frame. If *frame* is the selected frame, this makes *window* the selected window. This function returns *window*.

Conversely, selecting a window for Emacs with **select-window** also makes that window selected within its frame. See Section 28.4 [Selecting Windows], page 502.

Another function that (usually) returns one of the windows in a given frame is **minibuffer-window**. See [Definition of minibuffer-window], page 300.

## 29.8 Minibuffers and Frames

Normally, each frame has its own minibuffer window at the bottom, which is used whenever that frame is selected. If the frame has a minibuffer, you can get it with **minibuffer-window** (see [Definition of minibuffer-window], page 300).

However, you can also create a frame with no minibuffer. Such a frame must use the minibuffer window of some other frame. When you create the frame, you can specify explicitly the minibuffer window to use (in some other frame). 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 29.11 [Raising and Lowering], page 546.

**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 29.2 [Multiple Displays], page 530.

## 29.9 Input Focus

At any time, one frame in Emacs is the *selected frame*. The selected window always resides on the selected frame.

When Emacs displays its frames on several terminals (see Section 29.2 [Multiple Displays], page 530), 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 displayed 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 frame has the focus. To switch to a different frame from a Lisp function, call **select-frame-set-input-focus**.

Lisp programs can also 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-only 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` or `select-frame-set-input-focus`. Each terminal frame has a number which appears in the mode line before the buffer name (see Section 23.4.4 [Mode Line Variables], page 405).

**select-frame-set-input-focus** *frame*

[Function]

This function makes *frame* the selected frame, raises it (should it happen to be obscured by other frames) and tries to give it the X server’s focus. On a text-only terminal, the next redisplay displays the new frame on the entire terminal screen. The return value of this function is not significant.

**select-frame** *frame*

[Function]

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, as explained above, and the terminal that *frame* is on becomes the selected terminal. 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. It does so by generating a special kind of input event, called a *focus* event, when appropriate. The command loop handles a focus event by calling `handle-switch-frame`. See Section 21.6.9 [Focus Events], page 322.

**handle-switch-frame** *frame*

[Command]

This function handles a focus event by selecting frame *frame*.

Focus events normally do their job by invoking this command. Don’t call it for any other reason.

**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.

#### `focus-follows-mouse`

[User Option]

This option is how you inform Emacs whether the window manager transfers focus when the user moves the mouse. Non-`nil` says that it does. When this is so, the command `other-frame` moves the mouse to a position consistent with the new selected frame. (This option has no effect on MS-Windows, where the mouse pointer is always automatically moved by the OS to the selected frame.)

## 29.10 Visibility of Frames

A window frame may be *visible*, *invisible*, or *iconified*. If it is visible, you can see its contents, unless other windows cover it. If it is iconified, the frame's contents do not appear on the screen, but an icon does. If the frame is invisible, it doesn't show on the screen, not even as an icon.

Visibility is meaningless for terminal frames, since only the selected one is actually displayed in any case.

#### `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 29.11 [Raising and Lowering], page 546).

#### `make-frame-invisible &optional frame force`

[Command]

This function makes frame `frame` invisible. If you omit `frame`, it makes the selected frame invisible.

Unless `force` is non-`nil`, this function refuses to make `frame` invisible if all other frames are invisible..

#### `iconify-frame &optional frame`

[Command]

This function iconifies frame `frame`. If you omit `frame`, it iconifies the selected frame.

#### `frame-visible-p frame`

[Function]

This 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-only terminal, all frames are considered visible, whether they are currently being displayed or not, and this function returns `t` for all frames.

The visibility status of a frame is also available as a frame parameter. You can read or change it as such. See Section 29.3.3.6 [Management Parameters], page 536.

The user can 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 21.6.10 [Misc Events], page 322.

## 29.11 Raising and Lowering Frames

Most window systems use a desktop metaphor. Part of this metaphor is the idea that windows are stacked in a notional third dimension perpendicular to the screen surface, and thus ordered from “highest” to “lowest.” Where two windows overlap, the one higher up covers the one underneath. Even a window at the bottom of the stack can be seen if no other window overlaps it.

A window’s place in this ordering is not fixed; in fact, users tend to change the order frequently. *Raising* a window means moving it “up,” to the top of the stack. *Lowering* a window means moving it to the bottom of the stack. This motion is in the notional third dimension only, and does not change the position of the window on the screen.

You can raise and lower Emacs frame Windows with these functions:

**raise-frame &optional frame** [Command]

This function raises frame *frame* (default, the selected frame). If *frame* is invisible or iconified, this makes it visible.

**lower-frame &optional frame** [Command]

This function lowers frame *frame* (default, the selected frame).

**minibuffer-auto-raise** [User Option]

If this is non-*nil*, activation of the minibuffer raises the frame that the minibuffer window is in.

You can also enable auto-raise (raising automatically when a frame is selected) or auto-lower (lowering automatically when it is deselected) for any frame using frame parameters. See Section 29.3.3.6 [Management Parameters], page 536.

## 29.12 Frame Configurations

A *frame configuration* records the current arrangement of frames, all their properties, and the window configuration of each one. (See Section 28.18 [Window Configurations], page 526.)

**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.

## 29.13 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...* [Special Form]

This special form 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 21.6.8 [Motion Events], page 321, 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.

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 32.19.4 [Special Properties], page 620). That works at a much lower level and runs more smoothly than Lisp-level mouse tracking.

## 29.14 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 position in characters relative to the top left corner of the inside of *frame*.

**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.

**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 characters relative to the top left corner of the inside 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. These coordinates are not required to be within the frame.

If *frame* is not visible, this function does nothing. The return value is not significant.

## 29.15 Pop-Up Menus

When using a window system, a Lisp program can pop up a menu so that the user can choose an alternative with the mouse.

**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. 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 22.17 [Menu Keymaps], page 370). 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.

Alternatively, *menu* can have the following form:

`(title pane1 pane2...)`

where each pane is a list of form

`(title item1 item2...)`

Each item should normally be a cons cell (`(line . value)`), where *line* is a string, and *value* is the value to return if that *line* is chosen. An item can also be a string; this makes a non-selectable line in the menu.

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 keyboard input, 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 22.17.5 [Menu Bar], page 377.

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.

## 29.16 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 742).

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.

## 29.17 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 `text` (or `nil`), `arrow`, `hand`, `vdrag`, `hdrag`, `modeline`, and `hourglass`. `text` stands for the usual mouse pointer style used over text.

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`

[Variable]

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.

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 29.3.3.8 [Color Parameters], page 537.

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.

## 29.18 Window System Selections

The X server records a set of *selections* which permit transfer of data between application programs. The various selections are distinguished by *selection types*, represented in Emacs by symbols. X clients including Emacs can read or set the selection for any given type.

### `x-set-selection type data`

[Command]

This function sets a “selection” in the X server. It takes two arguments: a selection type `type`, and the value to assign to it, `data`. If `data` is `nil`, it means to clear out the selection. Otherwise, `data` may be a string, a symbol, an integer (or a cons of two integers or list of two integers), 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.

Each possible *type* has its own selection value, which changes independently. The usual values of *type* are PRIMARY, SECONDARY and CLIPBOARD; these are symbols with upper-case names, in accord with X Window System conventions. If *type* is nil, that stands for PRIMARY.

This function returns *data*.

**x-get-selection &optional type data-type** [Function]

This function accesses selections set up by Emacs or by other X clients. 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 X client 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.

The X server also has a set of eight numbered *cut buffers* which can store text or other data being moved between applications. Cut buffers are considered obsolete, but Emacs supports them for the sake of X clients that still use them. Cut buffers are numbered from 0 to 7.

**x-get-cut-buffer &optional n** [Function]

This function returns the contents of cut buffer number *n*. If omitted *n* defaults to 0.

**x-set-cut-buffer string &optional push** [Function]

This function stores *string* into the first cut buffer (cut buffer 0). If *push* is nil, only the first cut buffer is changed. If *push* is non-nil, that says to move the values down through the series of cut buffers, much like the way successive kills in Emacs move down the kill ring. In other words, the previous value of the first cut buffer moves into the second cut buffer, and the second to the third, and so on through all eight cut buffers.

**selection-coding-system** [Variable]

This variable specifies the coding system to use when reading and writing selections or the clipboard. See Section 33.10 [Coding Systems], page 648. 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. **x-get-selection** and **x-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.

On Mac OS, selection-like data transfer between applications is performed through a mechanism called *scraps*. The clipboard is a particular scrap named com.apple.scrap.clipboard. Types of scrap data are called *scrap flavor types*, which are identified by four-char codes such as TEXT. Emacs associates a selection with a scrap, and a selection type with a scrap flavor type via **mac-scrap-name** and **mac-ostype** properties, respectively.

```
(get 'CLIPBOARD 'mac-scrap-name)
     ⇒ "com.apple.scrap.clipboard"
(get 'com.apple.traditional-mac-plain-text 'mac-ostype)
     ⇒ "TEXT"
```

Conventionally, selection types for scrap flavor types on Mac OS have the form of UTI (Uniform Type Identifier) such as `com.apple.traditional-mac-plain-text`, `public.utf16-plain-text`, and `public.file-url`.

#### **x-select-enable-clipboard**

[User Option]

If this is non-`nil`, the Emacs yank functions consult the clipboard before the primary selection, and the kill functions store in the clipboard as well as the primary selection. Otherwise they do not access the clipboard at all. The default is `nil` on most systems, but `t` on MS-Windows and Mac.

## 29.19 Drag and Drop

When a user drags something from another application over Emacs, that other application expects Emacs to tell it if Emacs can handle the data that is dragged. The variable `x-dnd-test-function` is used by Emacs to determine what to reply. 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`. You can customize `x-dnd-test-function` and/or `x-dnd-known-types` if you want Emacs to accept or reject drops based on some other criteria.

If you want to change the way Emacs handles drop of different types or add a new type, customize `x-dnd-types-alist`. This requires detailed knowledge of what types other applications use for drag and drop.

When an URL is dropped on Emacs it may be a file, but it may also be another URL type (ftp, http, etc.). Emacs first checks `dnd-protocol-alist` to determine what to do with the URL. If there is no match there and if `browse-url-browser-function` is an alist, Emacs looks for a match there. If no match is found the text for the URL is inserted. If you want to alter Emacs behavior, you can customize these variables.

## 29.20 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 29.9 [Input Focus], page 543, for the meaning of the term “selected frame.”

**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")
  ⇒ (65280 65280 65280)
(color-values "red")
  ⇒ (65280 0 0)
(color-values "pink")
  ⇒ (65280 49152 51968)
(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.

## 29.21 Text Terminal Colors

Text-only 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 29.20 [Color Names], page 552.

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 more than one text-only terminal at one time; then this argument will specify which terminal to operate on (the default being the selected frame's terminal; see Section 29.9 [Input Focus], page 543). 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-only terminal.

**tty-color-alist** *&optional frame* [Function]

This function returns an alist recording the known colors supported by a text-only 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*.

## 29.22 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 application names other than “Emacs” 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"
```

See section “X Resources” in *The GNU Emacs Manual*.

## 29.23 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 29.9 [Input Focus], page 543).

See Section 29.20 [Color Names], page 552, Section 29.21 [Text Terminal Colors], page 554, 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 user cannot choose menu items without a mouse.

**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-only 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 38.12.2 [Face Attributes], page 765).

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 width for all physical monitors associated with *display*. See Section 29.2 [Multiple Displays], page 530.

**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 29.2 [Multiple Displays], page 530.

**display-mm-height &optional *display*** [Function]

This function returns the height of the screen in millimeters, or `nil` if Emacs cannot get that information.

**display-mm-width &optional *display*** [Function]

This function returns the width of the screen in millimeters, or `nil` if Emacs cannot get that information.

**display-mm-dimensions-alist** [Variable]

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`, `gray-scale`, `static-color`, `pseudo-color`, `true-color`, and `direct-color`.

**display-color-cells &optional *display*** [Function]

This function returns the number of color cells the screen supports.

These functions obtain additional information specifically about X displays.

**x-server-version &optional *display*** [Function]

This function returns the list of version numbers of the X server running the display. The value is a list of three integers: the major and minor version numbers of the X protocol, and the distributor-specific release number of the X server software itself.

**x-server-vendor &optional *display*** [Function]

This function returns the “vendor” that provided the X server software (as a string). Really this means whoever distributes the X server.

When the developers of X labelled software distributors as “vendors,” they showed their false assumption that no system could ever be developed and distributed non-commercially.

## 30 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 31 [Markers], page 572.

See also the “field” feature (see Section 32.19.11 [Fields], page 630), which provides functions that are used by many cursor-motion commands.

### 30.1 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 30.4 [Narrowing], page 569), 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 28.9 [Window Point], page 511, 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 30.4 [Narrowing], page 569.)

**point-max**

[Function]

This function returns the maximum accessible value of point in the current buffer. This is  $(1+ (\text{buffer-size}))$ , unless narrowing is in effect, in which case it is the position of the end of the region that you narrowed to. (See Section 30.4 [Narrowing], page 569.)

**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 30.4 [Narrowing], page 569), `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
```

## 30.2 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 30.1 [Point], page 559.

### 30.2.1 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 *position* is less than 1, it moves point to the beginning of the buffer. If *position* is greater than the length of the buffer, it moves point to the end.

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.

### 30.2.2 Motion by Words

These functions for parsing words use the syntax table to decide whether a given character is part of a word. See Chapter 35 [Syntax Tables], page 684.

**forward-word &optional count** [Command]

This function moves point forward *count* words (or backward if *count* is negative). If *count* is **nil**, it moves forward one word.

“Moving one word” means moving until point crosses a word-constituent character and then encounters a word-separator character. However, this function cannot move point past the boundary of the accessible portion of the buffer, or across a field boundary (see Section 32.19.11 [Fields], page 630). 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.

In an interactive call, *count* is specified by the numeric prefix argument. If *count* is omitted or **nil**, it defaults to 1.

**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** [Variable]

This variable affects the behavior of **forward-word** and everything that uses it. 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.

### 30.2.3 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!

### 30.2.4 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.

**goto-line line**

[Command]

This function moves point to the front of the *lineth* line, counting from line 1 at beginning of the buffer. If *line* is less than 1, it moves point to the beginning of the buffer. If *line* is greater than the number of lines in the buffer, it moves point to the end of the buffer—that is, the *end of the last line* of the buffer. This is the only case in which *goto-line* does not necessarily move to the beginning of a line.

If narrowing is in effect, then *line* still counts from the beginning of the buffer, but point cannot go outside the accessible portion. So *goto-line* moves point to the beginning or end of the accessible portion, if the line number specifies an inaccessible position.

The return value of *goto-line* is the difference between *line* and the line number of the line to which point actually was able to move (in the full buffer, before taking account of narrowing). Thus, the value is positive if the scan encounters the real end of the buffer before finding the specified line. The value is zero if scan encounters the end of the accessible portion but not the real end of the buffer.

In an interactive call, *line* is the numeric prefix argument if one has been provided. Otherwise *line* is read in the minibuffer.

**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 32.19.11 [Fields], page 630) 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 32.19.11 [Fields], page 630) 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.

**forward-line &optional count** [Command]

This function moves point forward *count* lines, to the beginning of the line. If *count* is negative, it moves point  $-count$  lines backward, to the beginning of a line. 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.

In an interactive call, *count* is the numeric prefix argument.

**count-lines start end** [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.

Here is an example of using `count-lines`:

```
(defun current-line ()
  "Return the vertical position of point..."
  (+ (count-lines (window-start) (point))
     (if (= (current-column) 0) 1 0)))
```

**line-number-at-pos** &optional pos [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.

Also see the functions `bolp` and `eolp` in Section 32.1 [Near Point], page 581. These functions do not move point, but test whether it is already at the beginning or end of a line.

### 30.2.5 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 38.3 [Truncation], page 740.

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 38.20 [Usual Display], page 806.

These functions scan text to determine where screen lines break, and thus take time proportional to the distance scanned. If you intend to use them heavily, Emacs provides caches which may improve the performance of your code. See Section 38.3 [Truncation], page 740.

**vertical-motion** count &optional window [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.

`vertical-motion` returns the number of screen lines over which it moved point. 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.

**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. 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).

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 window line number point has moved to, with the top line in the window numbered 0.

`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 20.12 [Minibuffer Contents], page 301.

### 30.2.6 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 35 [Syntax Tables], page 684. See Section 35.6 [Parsing Expressions], page 691, 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** [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.

**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 35.6 [Parsing Expressions], page 691. 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 argth 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 argth 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.

**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.

**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.

### 30.2.7 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 35.5 [Motion and Syntax], page 691.

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 34 [Searching and Matching], page 661).

**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 See Section 34.3 [Regular Expressions], page 663. Character classes can also be used, e.g. “[[:alnum:]]”. See see Section 34.3.1.2 [Char Classes], page 667.

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 ")
⇒ nil  
  

----- 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.

### 30.3 Excursions

It is often useful to move point “temporarily” within a localized portion of the program, or to switch buffers temporarily. This is called an *excursion*, and it is done with the **save-excursion** special form. This construct initially remembers the identity of the current buffer, and its values of point and the mark, and restores them after the completion of the excursion.

The forms for saving and restoring the configuration of windows are described elsewhere (see Section 28.18 [Window Configurations], page 526, and see Section 29.12 [Frame Configurations], page 546).

**save-excursion** *body...*

[Special Form]

The **save-excursion** special form saves the identity of the current buffer and the values of point and the mark in it, evaluates *body*, and finally restores the buffer and its saved values of point and the mark. All three saved values are restored even in case of an abnormal exit via **throw** or **error** (see Section 10.5 [Nonlocal Exits], page 125).

The **save-excursion** special form is the standard way to switch buffers or move point within one part of a program and avoid affecting the rest of the program. It is used more than 4000 times in the Lisp sources of Emacs.

**save-excursion** does not save the values of point and the mark for other buffers, so changes in other buffers remain in effect after **save-excursion** exits.

Likewise, **save-excursion** does not restore window-buffer correspondences altered by functions such as **switch-to-buffer**. One way to restore these correspondences, and the selected window, is to use **save-window-excursion** inside **save-excursion** (see Section 28.18 [Window Configurations], page 526).

The value returned by **save-excursion** is the result of the last form in *body*, or **nil** if no body forms were given.

```
(save-excursion forms)
≡
(let ((old-buf (current-buffer))
      (old-pnt (point-marker))
      (old-mark (copy-marker (mark-marker))))
  (unwind-protect
    (progn forms)
    (set-buffer old-buf)
    (goto-char old-pnt)
    (set-marker (mark-marker) old-mark)))
```

**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 31.5 [Marker Insertion Types], page 576. Therefore, when the saved point value is restored, it normally comes before the inserted text.

Although **save-excursion** saves the location of the mark, it does not prevent functions which modify the buffer from setting **deactivate-mark**, and thus causing the deactivation of the mark after the command finishes. See Section 31.7 [The Mark], page 577.

## 30.4 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 most Emacs 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.

Values such as positions or line numbers, which usually count from the beginning of the buffer, do so despite narrowing, but the functions which use them refuse to operate on text that is inaccessible.

The commands for saving buffers are unaffected by narrowing; they save the entire buffer regardless of any narrowing.

**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).

**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 34.8 [Standard Regexp], page 683).

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)))
```

**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 10.5 [Nonlocal Exits], page 125). Therefore, this construct is a clean way to narrow a buffer temporarily.

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 and the mark; 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 -----
```

# 31 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.

## 31.1 Overview of Markers

A marker specifies a buffer and a position in that buffer. The marker can be used to represent a position in the 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 30 [Positions], page 559, 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 often during the life of the marker. 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 31.5 [Marker Insertion Types], page 576)—unless the insertion is done with `insert-before-markers` (see Section 32.4 [Insertion], page 585).

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. Unreferenced markers are garbage collected eventually, but until then will continue to use time if they do point somewhere.

Because it is common to perform arithmetic operations on a marker position, most of the arithmetic 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>
```

## 31.2 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.

## 31.3 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 31.5 [Marker Insertion Types], page 576.

**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 30.1 [Point], page 559. 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 30.4 [Narrowing], page 569.

**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 30.4 [Narrowing], page 569.

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 15573 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** *marker-or-integer &optional 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 31.5 [Marker Insertion Types], page 576.

If passed an integer argument less than 1, `copy-marker` returns a new marker that points to the beginning of the current buffer. If passed an integer argument greater than the length of the buffer, `copy-marker` returns a new marker that points to the end of the buffer.

```
(copy-marker 0)
⇒ #<marker at 1 in markers.texi>
```

```
(copy-marker 20000)
⇒ #<marker at 7572 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
```

### 31.4 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
```

**buffer-has-markers-at** *position* [Function]  
 This function returns *t* if one or more markers point at position *position* in the current buffer.

## 31.5 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*.

Most functions that create markers, without an argument allowing to specify the insertion type, create them with insertion type `nil`. Also, the mark has, by default, insertion type `nil`.

## 31.6 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 less than 1, `set-marker` moves *marker* to the beginning of the buffer. If *position* is greater than the size of the buffer, `set-marker` moves *marker* to the end of the 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`.

## 31.7 The Mark

One special marker in each buffer is designated *the mark*. It specifies a position to bound a range of text for commands such as `kill-region` and `indent-rigidly`. Lisp programs should set the mark only to values that have 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.

Many commands are designed to operate on the text between point and the mark when called interactively. If you are writing such a command, 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 21.2.2 [Interactive Codes], page 307.

Each buffer has a marker which represents the value of the mark in that buffer, independent of any other buffer. When a buffer is newly created, this marker exists but does not point anywhere. That means the mark “doesn't exist” in that buffer as yet.

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 variable `mark-active`, which is always buffer-local in all buffers, indicates whether the mark is active: `non-nil` means yes. A command can request deactivation of the mark upon return to the editor command loop by setting `deactivate-mark` to a `non-nil` value (but this causes deactivation only if Transient Mark mode is enabled).

The main motivation for using Transient Mark mode is that this mode also enables highlighting of the region when the mark is active. See Chapter 38 [Display], page 739.

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 anyway (or `nil`).

### `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. `push-mark` returns `nil`.

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.

The return value is not meaningful.

**transient-mark-mode** [User Option]

This variable if non-`nil` enables Transient Mark mode, in which every buffer-modifying primitive sets `deactivate-mark`. The consequence of this is that commands that modify the buffer normally make the mark inactive.

Lisp programs can set `transient-mark-mode` to `only` to enable Transient Mark mode for the following command only. During that following command, the value of `transient-mark-mode` is `identity`. If it is still `identity` at the end of the command, it changes to `nil`.

**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.

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**

[Function]

This function deactivates the mark, if Transient Mark mode is enabled. 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.

**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 at the end of a command if the mark is active and it is possible that the region may have changed.

**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.

## 31.8 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.

Few programs need to use the `region-beginning` and `region-end` functions. 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 21.2.2 [Interactive Codes], page 307.)

## 32 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 32.9 [Undo], page 596).

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 31 [Markers], page 572) or numeric character positions (see Chapter 30 [Positions], page 559). 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.

### 32.1 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 34.4 [Regexp Search], page 673.

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 ‘@’:

```
(char-to-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*c)e! Peace!,’’
but there is no peace.
----- Buffer: foo -----
```

```
(char-to-string (preceding-char))
⇒ "a"
(char-to-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 30.1 [Point], page 559.

**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 30.1 [Point], page 559.

**bolp** [Function]

This function returns `t` if point is at the beginning of a line. See Section 30.2.4 [Text Lines], page 562. 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.

## 32.2 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.

It is not necessary for `start` to be less than `end`; the arguments can be given in either order. But most often the smaller argument is written first.

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 32.19 [Text Properties], page 615. However, overlays (see Section 38.9 [Overlays], page 754) 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 32.19 [Text Properties], page 615.

#### **filter-buffer-substring start end &optional delete noprops** [Function]

This function passes the buffer text between `start` and `end` through the filter functions specified by the variable `buffer-substring-filters`, and returns the value from the last filter function. If `buffer-substring-filters` is `nil`, the value is the unaltered text from the buffer, what `buffer-substring` would return.

If `delete` is non-`nil`, this function deletes the text between `start` and `end` after copying it, like `delete-and-extract-region`.

If `noprops` is non-`nil`, the final string returned does not include text properties, while the string passed through the filters still includes text properties from the buffer text.

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 add functions to `buffer-substring-filters` to alter such text as it is copied out of the buffer.

#### **buffer-substring-filters** [Variable]

This variable should be a list of functions that accept a single argument, a string, and return a string. `filter-buffer-substring` passes the buffer substring to the first function in this list, and the return value of each function is passed to the next function. The return value of the last function is used as the return value of `filter-buffer-substring`.

As a special convention, point is set to the start of the buffer text being operated on (i.e., the `start` argument for `filter-buffer-substring`) before these functions are called.

If this variable is `nil`, no filtering is performed.

**buffer-string** [Function]

This function returns the contents of the entire accessible portion of the current buffer as a string. It is equivalent to

```
(buffer-substring (point-min) (point-max))
----- Buffer: foo -----
This is the contents of buffer foo

----- Buffer: foo -----
```

```
(buffer-string)
⇒ "This is the contents of buffer foo\n"
```

**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*** [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`, `url`, `word`, `sentence`, `whitespace`, `line`, `page`, and others.

```
----- 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
```

### 32.3 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 the current buffer contains the text ‘foobarbar haha!rara!'; 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
```

## 32.4 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 relocates markers that point at positions after the insertion point, so that they stay with the surrounding text (see Chapter 31 [Markers], page 572). 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 31.5 [Marker Insertion Types], page 576). 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 or if they insert within read-only text.

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 33.2 [Converting Representations], page 641.

**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 count &optional inherit* [Function]

This function inserts *count* instances of *character* into the current buffer before point. The argument *count* should be an integer, and *character* must be a character. The value is **nil**.

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 33.2 [Converting Representations], page 641.

If *inherit* is non-**nil**, then the inserted characters inherit sticky text properties from the two characters before and after the insertion point. See Section 32.19.6 [Sticky Properties], page 625.

**insert-buffer-substring** *from-buffer-or-name &optional start end* [Function]

This function inserts a portion of buffer *from-buffer-or-name* (which must already exist) into the current buffer before point. The text inserted is the region between *start* and *end*. (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 truths*
----- 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.

See Section 32.19.6 [Sticky Properties], page 625, 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.

## 32.5 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* [Command]

This command inserts 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 33.9 [Translation of Characters], page 647.

This command calls **auto-fill-function** whenever that is non-`nil` and the character inserted is in the table **auto-fill-chars** (see Section 32.14 [Auto Filling], page 604).

This command performs abbrev expansion if Abbrev mode is enabled and the inserted character does not have word-constituent syntax. (See Chapter 36 [Abbrevs], page 699, and Section 35.2.1 [Syntax Class Table], page 685.) It is also responsible for calling **blink-paren-function** when the inserted character has close parenthesis syntax (see Section 38.19 [Blinking], page 805).

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* [Command]

This command inserts newlines into the current buffer before point. If *number-of-newlines* is supplied, that many newline characters are inserted.

This function calls **auto-fill-function** if the current column number is greater than the value of **fill-column** and *number-of-newlines* is `nil`. 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 indents to the left margin if that is not zero. See Section 32.12 [Margins], page 602.

The value returned is `nil`. In an interactive call, *count* is the numeric prefix argument.

**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).

## 32.6 Deleting Text

Deletion means removing part of the text in a buffer, without saving it in the kill ring (see Section 32.8 [The Kill Ring], page 591). Deleted text can't be yanked, but can be reinserted

using the undo mechanism (see Section 32.9 [Undo], page 596). 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.

## 32.7 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* [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. 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 32.12 [Margins], page 602.

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 35.2.1 [Syntax Class Table], page 685.

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**.

## 32.8 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 32.6 [Deletion], page 587.

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.

### 32.8.1 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.

### 32.8.2 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.

`kill-region start end &optional yank-handler` [Command]

This function kills the text in the region defined by `start` and `end`. 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.

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.

If `yank-handler` is non-`nil`, this puts that value onto the string of killed text, as a `yank-handler` text property. See Section 32.8.3 [Yanking], page 592. Note that if `yank-handler` is `nil`, any `yank-handler` properties present on the killed text are copied onto the kill ring, like other text properties.

`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` [Command]

This command saves the region defined by `start` and `end` on the kill ring (including text properties), but does not delete the text from the buffer. It returns `nil`.

The command does not set `this-command` to `kill-region`, so a subsequent kill command does not append to the same kill ring entry.

Don’t call `copy-region-as-kill` in Lisp programs unless you aim to support Emacs 18. For newer Emacs versions, it is better to use `kill-new` or `kill-append` instead. See Section 32.8.5 [Low-Level Kill Ring], page 594.

### 32.8.3 Yanking

Yanking means inserting text from the kill ring, but it does not insert the text blindly. Yank commands and some other commands use `insert-for-yank` to perform special processing on the text that they copy into the buffer.

`insert-for-yank string` [Function]

This function normally works like `insert` except that it doesn’t insert the text properties in the `yank-excluded-properties` list. However, if any part of `string` has a non-`nil` `yank-handler` text property, that property can do various special processing on that part of the text being inserted.

`insert-buffer-substring-as-yank buf &optional start end` [Function]

This function resembles `insert-buffer-substring` except that it doesn’t insert the text properties in the `yank-excluded-properties` list.

You can put a `yank-handler` text property on all or part of the text to control how it will be inserted if it is yanked. The `insert-for-yank` function looks for that property. 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 present and non-`nil`, it is called instead of `insert` to insert the string. `function` takes one argument—the string to insert.
- `param` If `param` is present and non-`nil`, it replaces `string` (or the part 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`, the normal removal of the yank-excluded-properties is not performed; instead `function` is responsible for removing those properties. This may be necessary if `function` adjusts point before or after inserting the object.
- `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.

### 32.8.4 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 32.8.3 [Yanking], page 592).

`yank` &optional arg

[Command]

This command inserts before point the text at the front of the kill ring. It positions the mark at the beginning of that text, and 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 puts 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]

This command replaces the just-yanked entry from the kill ring with a different entry from the kill ring.

This is allowed only immediately after a `yank` or another `yank-pop`. At such a time, the region contains text that was just inserted by yanking. `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 that after the oldest one comes the newest one, and before the newest one goes the oldest.

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.

### 32.8.5 Low-Level Kill Ring

These functions and variables provide access to the kill ring at a lower level, but still convenient for use in Lisp programs, because they take care of interaction with window system selections (see Section 29.18 [Window System Selections], page 550).

#### `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, `current-kill` pushes that string onto the kill ring and returns it. It also sets the yanking pointer to point to that new entry, 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 yank-handler` [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 value of `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.

If `yank-handler` is non-`nil`, this puts that value onto the string of killed text, as a `yank-handler` property. See Section 32.8.3 [Yanking], page 592. Note that if `yank-handler` is `nil`, then `kill-new` copies any `yank-handler` properties present on `string` onto the kill ring, as it does with other text properties.

**`kill-append` *string before-p* &optional *yank-handler***

[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 also invokes the value of `interprogram-cut-function` (see below). This handles `yank-handler` just like `kill-new`, except that if `yank-handler` is different from the `yank-handler` property of the first entry of the kill ring, `kill-append` pushes the concatenated string onto the kill ring, instead of replacing the original first entry with it.

**`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.

The normal use of this hook is to get the window system’s primary selection as the most recent kill, even if the selection belongs to another application. See Section 29.18 [Window System Selections], page 550.

**`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 and one optional 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 first argument. The second, optional, argument has the same meaning as the `push` argument to `x-set-cut-buffer` (see [Definition of `x-set-cut-buffer`], page 551) and only affects the second and later cut buffers.

The normal use of this hook is to set the window system’s primary selection (and first cut buffer) from the newly killed text. See Section 29.18 [Window System Selections], page 550.

### 32.8.6 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.

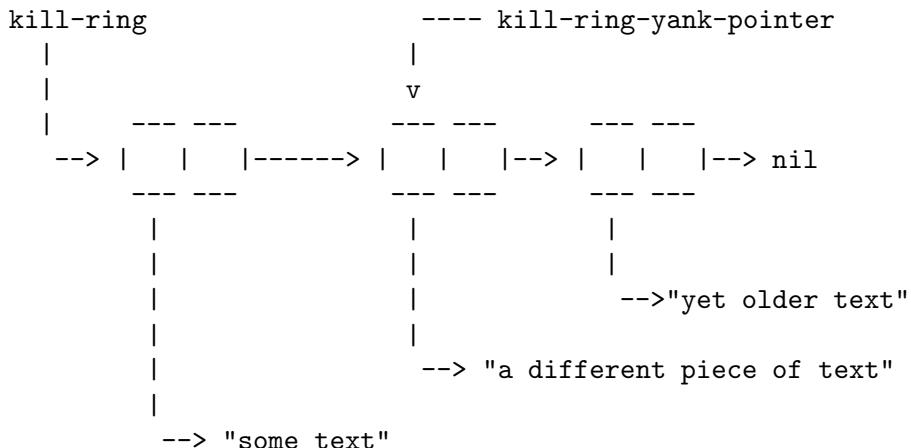
The `kill-ring-yank-pointer` variable points to a link in the kill ring list, whose CAR is the text to yank next. We say it identifies the “front” of the ring. Moving `kill-ring-yank-pointer` to a different link is called *rotating the kill ring*. We call the kill ring a “ring” because the functions 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-v* 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 60.

### 32.9 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 27.3 [Buffer Names], page 484.) 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.

**(*t high . low*)**

This kind of element indicates that an unmodified buffer became modified. The elements *high* and *low* are two integers, each recording 16 bits of the visited file's modification time as of when it was previously visited or saved. **primitive-undo** uses those values to determine whether to mark the buffer as unmodified once again; it does so only if the file's modification time matches those numbers.

**(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. 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*. 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**.

The editor command loop automatically creates an undo boundary before each key sequence is executed. Thus, each undo normally undoes the effects of one command. Self-inserting input characters are an exception. The command loop makes a boundary for the first such character; the next 19 consecutive self-inserting input characters do not make boundaries, and then the 20th does, and so on as long as self-inserting characters continue.

All buffer modifications add a boundary whenever the previous undoable change was made in some other buffer. This is to ensure that each command makes a boundary in each buffer where it makes changes.

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.

**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`.

## 32.10 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.

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.

## 32.11 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 32.14 [Auto Filling], page 604) 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 32.12 [Margins], page 602). 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 *justify***

[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*.

**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 34.8 [Standard Regexp], page 683.

**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 don’t canonicalize spaces before that position.

In Adaptive Fill mode, this command calls `fill-context-prefix` to choose a fill prefix by default. See Section 32.13 [Adaptive Fill], page 603.

**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 do 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-paragraph-function** [Variable]

This variable provides a way for major modes to override the filling of paragraphs. If the 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))
```

**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.

## 32.12 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 32.14 [Auto Filling], page 604).

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.

**default-fill-column** [Variable]

The value of this variable is the default value for *fill-column* in buffers that do not override it. This is the same as (*default-value* ‘*fill-column*’).

The default value for *default-fill-column* is 70.

**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** [Variable]

This variable specifies the base left margin column. In Fundamental mode, `C-j` indents to this column. This variable automatically becomes buffer-local when set in any fashion.

**fill-nobreak-predicate** [Variable]

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.

### 32.13 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 32.11 [Filling], page 599, and in Section 32.14 [Auto Filling], page 604.

**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.

## 32.14 Auto Filling

Auto Fill mode is a minor mode that fills lines automatically as text is inserted. This section describes the hook used by Auto Fill mode. For a description of functions that you can call explicitly to fill and justify existing text, see Section 32.11 [Filling], page 599.

Auto Fill mode also enables the functions that change the margins and justification style to refill portions of the text. See Section 32.12 [Margins], page 602.

**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`. 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.

In older Emacs versions, this variable was named `auto-fill-hook`, but since it is not called with the standard convention for hooks, it was renamed to `auto-fill-function` in version 19.

**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.

## 32.15 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 75). 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. If keys are numbers, it defaults to `<`; otherwise it defaults to `string<`.

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
  (function
    (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 34.3 [Regular Expressions], page 663, 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.

## 32.16 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 38.20 [Usual Display], page 806.

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.

### `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.

For an example of using `current-column`, see the description of `count-lines` in Section 30.2.4 [Text Lines], page 562.

### `move-to-column` *column* &`optional` *force*

[Function]

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.

If column *column* is beyond the end of the line, point moves to the end of the line. If *column* is negative, point moves to the beginning of the line.

If it is impossible to move to column *column* because that is in the middle of a multi-column 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` converts the tab into spaces so that it can move precisely to column *column*. Other multi-column 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.

If *column* is not an integer, an error is signaled.

The return value is the column number actually moved to.

## 32.17 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.

### 32.17.1 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 38.10 [Width], page 760, 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.

#### `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 32.19.6 [Sticky Properties], page 625.

#### `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.

### 32.17.2 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-line-function`

[Variable]

This variable's value is the function to be used by TAB (and various commands) to indent the current line. The command `indent-according-to-mode` does no more than call this function.

In Lisp mode, the value is the symbol `lisp-indent-line`; in C mode, `c-indent-line`; in Fortran mode, `fortran-indent-line`. The default value is `indent-relative`.

#### `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.

#### `indent-for-tab-command`

[Command]

This command calls the function in `indent-line-function` to indent the current line; however, if that function is `indent-to-left-margin`, `insert-tab` is called instead. (That is a trivial command that inserts a tab character.)

#### `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 the current `indent-line-function`. In programming language modes, this is the same thing TAB does, but in some text modes, where TAB inserts a tab, `newline-and-indent` indents to the column specified by `left-margin`.

**`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).

This command does indentation on both lines according to the current major mode, by calling the current value of `indent-line-function`. In programming language modes, this is the same thing TAB does, but in some text modes, where TAB inserts a tab, `reindent-then-newline-and-indent` indents to the column specified by `left-margin`.

### 32.17.3 Indenting an Entire Region

This section describes commands that indent all the lines in the region. They return unpredictable values.

**`indent-region start end 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 command 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. Consequently, this command 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 each of the lines beginning in the region specified.

In Mail mode, **C-c C-y** (`mail-yank-original`) uses `indent-rigidly` to indent the text copied from the message being replied to.

**indent-code-rigidly** *start end columns &optional nochange-regexp* [Function]

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*).

### 32.17.4 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 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.

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-maybe** [Command]

This command indents the current line like the previous nonblank line, by calling `indent-relative` with `t` as the *unindented-ok* argument. The return value is unpredictable.

If the previous nonblank line has no indent points beyond the current column, this command does nothing.

### 32.17.5 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 38.20 [Usual Display], page 806). 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`. It searches the list for an element greater than the current column number, and uses that element as the column to indent to. It does nothing if no such element is found.

#### `tab-stop-list`

[User Option]

This variable is the list of tab stop columns used by `tab-to-tab-stops`. The elements should be integers in increasing order. The tab stop columns need not be evenly spaced.

Use `M-x edit-tab-stops` to edit the location of tab stops interactively.

### 32.17.6 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). It returns `nil`.

#### `backward-to-indentation &optional arg`

[Command]

This command moves point backward `arg` lines and then to the first nonblank character on that line. It returns `nil`. 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. It returns `nil`. If `arg` is omitted or `nil`, it defaults to 1.

### 32.18 Case Changes

The case change commands described here work on text in the current buffer. See Section 4.8 [Case Conversion], page 58, for case conversion functions that work on strings and characters. See Section 4.9 [Case Tables], page 60, 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 44)
⇒ 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.

## 32.19 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 8.4 [Property Lists], page 107). 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 32.19.4 [Special Properties], page 620). 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`.

### 32.19.1 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 32.19.3 [Property Search], page 618, 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.

**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 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 38.9 [Overlays], page 754.

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 all overlays in that buffer are considered, as well as text properties. If *object* is a string, only text properties are considered, since strings never have overlays.

**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.

**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
```

### 32.19.2 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 32.9 [Undo], page 596). 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 8.4 [Property Lists], page 107): 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 8.4 [Property Lists], page 107): 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 &optional object* [Function]

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.

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* which has the text properties *properties*. 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 also the function `buffer-substring-no-properties` (see Section 32.2 [Buffer Contents], page 582) which copies text from the buffer but does not copy its properties.

### 32.19.3 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 high 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` till 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, `next-property-change` 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.

### 32.19.4 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 21.5 [Adjusting Point], page 315.

**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** You can use the property `face` to control the font and color of text. See Section 38.12 [Faces], page 762, for more information.

In the simplest case, the value is a face name. It can also be a list; then each element can be any of these possibilities;

- A face name (a symbol or string).
- A property list of face attributes. This has the form `(keyword value ...)`, where each `keyword` is a face attribute name and `value` is a meaningful value for that attribute. With this feature, you do not need to create a face each time you want to specify a particular attribute for certain text. See Section 38.12.2 [Face Attributes], page 765.
- A cons cell with the form `(foreground-color . color-name)` or `(background-color . color-name)`. These elements specify just the foreground color or just the background color. See Section 29.20 [Color Names], page 552, for the supported forms of `color-name`.

A cons cell of `(foreground-color . color-name)` is equivalent to specifying `(:foreground color-name)`; likewise for the background.

You can use Font Lock Mode (see Section 23.6 [Font Lock Mode], page 412), to dynamically update `face` properties based on the contents of the text.

**font-lock-face**

The **font-lock-face** property is the same in all respects as the **face** property, but its state of activation is controlled by **font-lock-mode**. This can be advantageous for special buffers which are not intended to be user-editable, or for static areas of text which are always fontified in the same way. See Section 23.6.6 [Precalculated Fontification], page 419.

Strictly speaking, **font-lock-face** is not a built-in text property; rather, it is implemented in Font Lock mode using **char-property-alist**. See Section 32.19.1 [Examining Properties], page 615.

This property is new in Emacs 22.1.

**mouse-face**

The property **mouse-face** is used instead of **face** when the mouse is on or near the character. For this purpose, “near” means that all text between the character and where the mouse is have the same **mouse-face** property value.

**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 38.12.7 [Auto Faces], page 773) 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 38.15 [Display Property], page 783.

**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 “Tooltips” in *The GNU Emacs Manual*).

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 624).

This feature is used in the mode line and for other active text.

<b>keymap</b>	The <b>keymap</b> 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 22.7 [Active Keymaps], page 353. 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 <b>non-nil</b> and rear-sticky, and the property's value for the character after point applies if it is <b>non-nil</b> and front-sticky. (For mouse clicks, the position of the click is used instead of the position of point.)
<b>local-map</b>	This property works like <b>keymap</b> except that it specifies a keymap to use <i>instead of</i> the buffer's local map. For most purposes (perhaps all purposes), it is better to use the <b>keymap</b> property.
<b>syntax-table</b>	The <b>syntax-table</b> property overrides what the syntax table says about this particular character. See Section 35.4 [Syntax Properties], page 690.
<b>read-only</b>	If a character has the property <b>read-only</b> , then modifying that character is not allowed. Any command that would do so gets an error, <b>text-read-only</b> . 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 <b>read-only</b> property due to stickiness. Thus, you can control permission to insert next to read-only text by controlling the stickiness. See Section 32.19.6 [Sticky Properties], page 625.  Since changing properties counts as modifying the buffer, it is not possible to remove a <b>read-only</b> property unless you know the special trick: bind <b>inhibit-read-only</b> to a <b>non-nil</b> value and then remove the property. See Section 27.7 [Read Only Buffers], page 489.
<b>invisible</b>	A <b>non-nil</b> <b>invisible</b> property can make a character invisible on the screen. See Section 38.6 [Invisible Text], page 748, for details.
<b>intangible</b>	If a group of consecutive characters have equal and <b>non-nil</b> <b>intangible</b> 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 <b>non-nil</b> <b>intangible</b> properties, they belong to separate groups; each group is separately treated as described above. When the variable <b>inhibit-point-motion-hooks</b> is <b>non-nil</b> , the <b>intangible</b> property is ignored.
<b>field</b>	Consecutive characters with the same <b>field</b> property constitute a <i>field</i> . Some motion functions including <b>forward-word</b> and <b>beginning-of-line</b> stop moving at a field boundary. See Section 32.19.11 [Fields], page 630.

**cursor** Normally, the cursor is displayed at the end of any overlay and text property strings present at the current window position. You can place the cursor on any desired character of these strings by giving that character a non-*nil* `cursor` text property.

**pointer** This specifies a specific pointer shape when the mouse pointer is over this text or image. See Section 29.17 [Pointer Shape], page 550, 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 38.11 [Line Height], page 761.

#### **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 38.11 [Line Height], page 761.

#### **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. 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.

If these functions modify the buffer, they should bind `inhibit-modification-hooks` to `t` around doing so, to avoid confusing the internal mechanism that calls these hooks.

Overlays also support the `modification-hooks` property, but the details are somewhat different (see Section 38.9.2 [Overlay Properties], page 756).

#### **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.

See also Section 32.26 [Change Hooks], page 638, 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 with `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.

**inhibit-point-motion-hooks**

[Variable]

When this 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`.

**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 22.17.1.1 [Simple Menu Items], page 371, see Section 22.17.1.2 [Extended Menu Items], page 372), or tool bar help strings (see Section 22.17.6 [Tool Bar], page 378). The specified function is called with one argument, the help string to display. Tooltip mode (see section “Tooltips” in *The GNU Emacs Manual*) provides an example.

**composition**

This text property is used to display a sequence of characters as a single glyph composed from components. For instance, in Thai a base consonant is composed with the following combining vowel as a single glyph. The value should be a character or a sequence (vector, list, or string) of integers.

- If it is a character, it means to display that character instead of the text in the region.
- If it is a string, it means to display that string’s contents instead of the text in the region.
- If it is a vector or list, the elements are characters interleaved with internal codes specifying how to compose the following character with the previous one.

### 32.19.5 Formatted Text Properties

These text properties affect the behavior of the fill commands. They are used for representing formatted text. See Section 32.11 [Filling], page 599, and Section 32.12 [Margins], page 602.

**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.

### 32.19.6 Stickiness of Text Properties

Self-inserting characters normally take on the same properties as the preceding character. This is called *inheritance* of properties.

In a Lisp program, you can do insertion with inheritance or without, depending on your 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 32.4 [Insertion], page 585, for the ordinary insertion functions which do not inherit.

### 32.19.7 Saving Text Properties in Files

You can save text properties in files (along with the text itself), and restore the same text properties when visiting or inserting the files, using these two hooks:

**write-region-annotate-functions** [Variable]

This variable's value is a list of functions for **write-region** to run to encode text properties in some fashion as annotations to the text being written in the file. See Section 25.4 [Writing to Files], page 441.

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 lists indicating annotations to write in the file in addition to the text in the buffer.

Each function should return a list of elements of the form (*position* . *string*), where *position* is an integer specifying the relative position within the text to be written, and *string* is the annotation to add there.

Each list returned by one of these functions must be already sorted in increasing order by *position*. If there is more than one function, **write-region** merges the lists destructively into one sorted list.

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.

**after-insert-file-functions** [Variable]

This variable holds a list of functions for **insert-file-contents** to call after inserting a file's contents. These functions should scan the inserted text for annotations, and convert them to the text properties they stand for.

Each function receives one argument, the length of the inserted text; point indicates the start of that text. The function should scan that text for annotations, delete

them, and create the text properties that the annotations specify. The function should return the updated length of the inserted text, as it stands after those changes. The value returned by one function becomes the argument to the next function.

These functions should always return with point at the beginning of the inserted text.

The intended use of `after-insert-file-functions` is for converting some sort of textual annotations into actual text properties. But other uses may be possible.

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.

See Section 25.12 [Format Conversion], page 468, for a related feature.

### 32.19.8 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.

### 32.19.9 Defining Clickable Text

*Clickable text* is text that can be clicked, with either the mouse or via keyboard commands, to produce some result. Many major modes use clickable text to implement features such as hyper-links. The `button` package provides an easy way to insert and manipulate clickable text. See Section 38.17 [Buttons], page 796.

In this section, we will explain how to manually set up clickable text in a buffer using text properties. This involves two things: (1) indicating clickability when the mouse moves over the text, and (2) making *RET* or a mouse click on that text do something.

Indicating clickability usually involves highlighting the text, and often involves displaying helpful information about the action, such as which mouse button to press, or a short summary of the action. This can be done with the `mouse-face` and `help-echo` text properties. See Section 32.19.4 [Special Properties], page 620. Here is an example of how Dired does it:

```
(condition-case nil
  (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")))
  (error nil))
```

The first two arguments to `add-text-properties` specify the beginning and end of the text.

The usual way to make the mouse do something when you click it on this text is to define `mouse-2` in the major mode's keymap. The job of checking whether the click was on clickable text is done by the command definition. Here is how Dired does it:

```
(defun dired-mouse-find-file-other-window (event)
  "In Dired, visit the file or directory name you click on."
  (interactive "e")
  (let (window pos file)
    (save-excursion
      (setq window (posn-window (event-end event)))
      pos (posn-point (event-end event)))
      (if (not (windowp window))
          (error "No file chosen"))
      (set-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)))))
```

The reason for the `save-excursion` construct is to avoid changing the current buffer. In this case, Dired uses the functions `posn-window` and `posn-point` to determine which buffer the

click happened in and where, and in that buffer, `direc-get-file-for-visit` to determine which file to visit.

Instead of defining a mouse command for the major mode, you can define a key binding for the clickable text itself, using the `keymap` text property:

```
(let ((map (make-sparse-keymap)))
  (define-key map [mouse-2] 'operate-this-button)
  (put-text-property (point)
    (save-excursion
      (direc-move-to-end-of-filename)
      (point))
    'keymap map))
```

This method makes it possible to define different commands for various clickable pieces of text. Also, the major mode definition (or the global definition) remains available for the rest of the text in the buffer.

### 32.19.10 Links and Mouse-1

The normal Emacs command for activating text in read-only buffers is MOUSE-2, which includes following textual links. However, most graphical applications use MOUSE-1 for following links. For compatibility, MOUSE-1 follows links in Emacs too, when you click on a link quickly without moving the mouse. The user can customize this behavior through the variable `mouse-1-click-follows-link`.

To define text as a link at the Lisp level, you should bind the `mouse-2` event to a command to follow the link. Then, to indicate that MOUSE-1 should also follow the link, you should specify a `follow-link` condition either as a text property or as a key binding:

#### `follow-link` property

If the clickable text has a non-`nil` `follow-link` text or overlay property, that specifies the condition.

#### `follow-link` event

If there is a binding for the `follow-link` event, either on the clickable text or in the local keymap, the binding is the condition.

Regardless of how you set the `follow-link` condition, its value is used as follows to determine whether the given position is inside a link, and (if so) to compute an *action code* saying how MOUSE-1 should handle the link.

#### `mouse-face`

If the condition is `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:

```
(define-key Info-mode-map [follow-link] 'mouse-face)
```

a function If the condition is a valid 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:

```
(define-key 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 only specify this kind of condition on the text that constitutes a link.

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 with a value as specified above to determine how to follow the link. 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 with a value as specified above to determine how to follow the link.

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 21.6.13 [Accessing Events], page 326).

### 32.19.11 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 32.19.6 [Sticky Properties], page 625). 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 30.4 [Narrowing], page 569.

**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* [Function]

*only-in-line* *inhibit-capture-property*

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 as well as returning it.

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 is 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.

### 32.19.12 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 “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. One 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 this character?” 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 32.19.3 [Property Search], page 618.

Emacs also provides explicit intervals as a presentation feature; see Section 38.9 [Overlays], page 754.

## 32.20 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*.

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 38.7 [Selective Display], page 750).

**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 -----
```

**translate-region** *start end table* [Function]

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 uchar*) gives the translated character corresponding to *uchar*. 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.

## 32.21 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.

### (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.

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. You can pass a non-*nil* second argument *beforep* to this function interactively by supplying any prefix argument.

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.

## 32.22 Transposition of Text

This subroutine is used by the transposition commands.

**transpose-regions** *start1 end1 start2 end2 &optional leave-markers* [Function]

This function exchanges two nonoverlapping portions of the buffer. 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.

## 32.23 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<sup>1</sup>2045. This section describes the functions for converting to and from this code.

**base64-encode-region** *beg end &optional no-line-break* [Function]

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 characters from the charsets **ascii**, **eight-bit-control** and **eight-bit-graphic**.

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.

**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.

---

<sup>1</sup> 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.

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.

**base64-decode-region** *beg end* [Function]

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.

**base64-decode-string** *string* [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.

## 32.24 MD5 Checksum

MD5 cryptographic checksums, or *message digests*, are 128-bit “fingerprints” of a document or program. They are used to verify that you have an exact and unaltered copy of the data. The algorithm to calculate the MD5 message digest is defined in Internet RFC<sup>2</sup>1321. This section describes the Emacs facilities for computing message digests.

**md5** *object &optional start end coding-system noerror* [Function]

This function returns the MD5 message digest of *object*, which should be a buffer or a string.

The two 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 digest is computed for the whole of *object*.

The function **md5** does not compute the message digest directly from the internal Emacs representation of the text (see Section 33.1 [Text Representations], page 640). Instead, it encodes the text using a coding system, and computes the message digest from the encoded text. The optional fourth argument *coding-system* specifies which coding system to use for encoding the text. It should be the same coding system that you used to read the text, or that you used or will use when saving or sending the text. See Section 33.10 [Coding Systems], page 648, for more information about coding systems.

If *coding-system* is *nil* or omitted, the default depends on *object*. If *object* is a buffer, the default for *coding-system* is whatever coding system would be chosen by default for writing this text into a file. If *object* is a string, the user’s most preferred coding system (see section “the description of **prefer-coding-system**” in *GNU Emacs Manual*) is used.

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.

---

<sup>2</sup> For an explanation of what is an RFC, see the footnote in Section 32.23 [Base 64], page 635.

## 32.25 Atomic Change Groups

In data base 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*.

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.

## 32.26 Change Hooks

These hook variables let you arrange to take notice of all changes in all buffers (or in a particular buffer, if you make them buffer-local). See also Section 32.19.4 [Special Properties], page 620, 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 before any buffer modification. 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.

### `after-change-functions`

[Variable]

This variable holds a list of functions to call after any buffer modification. 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's about to change is always the current buffer.

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.

### `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.

The two variables above are temporarily bound to `nil` during the time that any of these functions is running. This means that if one of these functions changes the buffer, that change won't run these functions. If you do want a hook function to make changes that run these functions, make it bind these variables back to their usual values.

One inconvenient result of this protective feature is that you cannot have a function in `after-change-functions` or `before-change-functions` which changes the value of that variable. But that's not a real limitation. If you want those functions to change the list of functions to run, simply add one fixed function to the hook, and code that function to look in another variable for other functions to call. Here is an example:

```
(setq my-own-after-change-functions nil)
(defun indirect-after-change-function (beg end len)
  (let ((list my-own-after-change-functions))
    (while list
      (funcall (car list) beg end len)
      (setq list (cdr list)))))

(add-hooks 'after-change-functions
           'indirect-after-change-function)
```

**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 32.19.4 [Special Properties], page 620) and overlay properties (see Section 38.9.2 [Overlay Properties], page 756).

## 33 Non-ASCII Characters

This chapter covers the special issues relating to non-ASCII characters and how they are stored in strings and buffers.

### 33.1 Text Representations

Emacs has two *text representations*—two ways to represent text in a string or buffer. These are called *unibyte* and *multibyte*. Each string, and each buffer, uses one of these two representations. For most purposes, you can ignore the issue of representations, because Emacs converts text between them as appropriate. Occasionally in Lisp programming you will need to pay attention to the difference.

In unibyte representation, each character occupies one byte and therefore the possible character codes range from 0 to 255. Codes 0 through 127 are ASCII characters; the codes from 128 through 255 are used for one non-ASCII character set (you can choose which character set by setting the variable `nonascii-insert-offset`).

In multibyte representation, a character may occupy more than one byte, and as a result, the full range of Emacs character codes can be stored. The first byte of a multibyte character is always in the range 128 through 159 (octal 0200 through 0237). These values are called *leading codes*. The second and subsequent bytes of a multibyte character are always in the range 160 through 255 (octal 0240 through 0377); these values are *trailing codes*.

Some sequences of bytes are not valid in multibyte text: for example, a single isolated byte in the range 128 through 159 is not allowed. But character codes 128 through 159 can appear in multibyte text, represented as two-byte sequences. All the character codes 128 through 255 are possible (though slightly abnormal) in multibyte text; they appear in multibyte buffers and strings when you do explicit encoding and decoding (see Section 33.10.7 [Explicit Encoding], page 656).

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 text.

You cannot set this variable directly; instead, use the function `set-buffer-multibyte` to change a buffer’s representation.

#### `default-enable-multibyte-characters`

[Variable]

This variable’s value is entirely equivalent to `(default-value 'enable-multibyte-characters)`, and setting this variable changes that default value. Setting the local binding of `enable-multibyte-characters` in a specific buffer is not allowed, but changing the default value is supported, and it is a reasonable thing to do, because it has no effect on existing buffers.

The ‘`--unibyte`’ command line option does its job by setting the default value to `nil` early in startup.

**position-bytes** *position* [Function]

Return 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 corresponding to byte-position *byte-position* in the current buffer. If *byte-position* is out of range, the value is **nil**.

**multibyte-string-p** *string* [Function]

Return **t** if *string* is a multibyte 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*).

## 33.2 Converting Text Representations

Emacs can convert unibyte text to multibyte; it can also convert multibyte text to unibyte, though this conversion loses information. 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 that it is constructed from. 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 likewise character codes 128 through 159. It converts the non-ASCII codes 160 through 255 by adding the value **nonascii-insert-offset** to each character code. By setting this variable, you specify which character set the unibyte characters correspond to (see Section 33.5 [Character Sets], page 644). For example, if **nonascii-insert-offset** is 2048, which is `(- (make-char 'latin-iso8859-1) 128)`, then the unibyte non-ASCII characters correspond to Latin 1. If it is 2688, which is `(- (make-char 'greek-iso8859-7) 128)`, then they correspond to Greek letters.

Converting multibyte text to unibyte is simpler: it discards all but the low 8 bits of each character code. If **nonascii-insert-offset** has a reasonable value, corresponding to the beginning of some character set, this conversion is the inverse of the other: converting unibyte text to multibyte and back to unibyte reproduces the original unibyte text.

**nonascii-insert-offset** [Variable]

This variable specifies the amount to add to a non-ASCII character when converting unibyte text to multibyte. It also applies when **self-insert-command** inserts a

character in the unibyte non-ASCII range, 128 through 255. However, the functions `insert` and `insert-char` do not perform this conversion.

The right value to use to select character set *cs* is (`- (make-char cs) 128`). If the value of `nonascii-insert-offset` is zero, then conversion actually uses the value for the Latin 1 character set, rather than zero.

**nonascii-translation-table**

[Variable]

This variable provides a more general alternative to `nonascii-insert-offset`. You can use it to specify independently how to translate each code in the range of 128 through 255 into a multibyte character. The value should be a char-table, or `nil`. If this is non-`nil`, it overrides `nonascii-insert-offset`.

The next three functions either return the argument *string*, or a newly created string with no text properties.

**string-make-unibyte** *string*

[Function]

This function converts the text of *string* to unibyte representation, if it isn't already, and returns the result. If *string* is a unibyte string, it is returned unchanged. Multibyte character codes are converted to unibyte according to `nonascii-translation-table` or, if that is `nil`, using `nonascii-insert-offset`. If the lookup in the translation table fails, this function takes just the low 8 bits of each character.

**string-make-multibyte** *string*

[Function]

This function converts the text of *string* to multibyte representation, if it isn't already, and returns the result. If *string* is a multibyte string or consists entirely of ASCII characters, it is returned unchanged. In particular, if *string* is unibyte and entirely ASCII, the returned string is unibyte. (When the characters are all ASCII, Emacs primitives will treat the string the same way whether it is unibyte or multibyte.) If *string* is unibyte and contains non-ASCII characters, the function `unibyte-char-to-multibyte` is used to convert each unibyte character to a multibyte character.

**string-to-multibyte** *string*

[Function]

This function returns a multibyte string containing the same sequence of character codes as *string*. Unlike `string-make-multibyte`, this function unconditionally returns a multibyte string. If *string* is a multibyte string, it is returned unchanged.

**multibyte-char-to-unibyte** *char*

[Function]

This convert the multibyte character *char* to a unibyte character, based on `nonascii-translation-table` and `nonascii-insert-offset`.

**unibyte-char-to-multibyte** *char*

[Function]

This convert the unibyte character *char* to a multibyte character, based on `nonascii-translation-table` and `nonascii-insert-offset`.

### 33.3 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; a sequence of two bytes which is treated as one character in multibyte representation will count as two characters in unibyte representation. Character codes 128 through 159 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.

You cannot use `set-buffer-multibyte` on an indirect buffer, because indirect buffers always inherit the representation of the base buffer.

**string-as-unibyte** *string*

[Function]

This function returns a string with the same bytes as *string* but treating each byte as a character. This means that the value may have more characters than *string* has.

If *string* is already a unibyte string, then the value is *string* itself. Otherwise it is a newly created string, with no text properties. If *string* is multibyte, any characters it contains of charset `eight-bit-control` or `eight-bit-graphic` are converted to the corresponding single byte.

**string-as-multibyte** *string*

[Function]

This function returns a string with the same bytes as *string* but treating each multi-byte sequence as one character. This means that the value may have fewer characters than *string* has.

If *string* is already a multibyte string, then the value is *string* itself. Otherwise it is a newly created string, with no text properties. If *string* is unibyte and contains any individual 8-bit bytes (i.e. not part of a multibyte form), they are converted to the corresponding multibyte character of charset `eight-bit-control` or `eight-bit-graphic`.

## 33.4 Character Codes

The unibyte and multibyte text representations use different character codes. The valid character codes for unibyte representation range from 0 to 255—the values that can fit in one byte. The valid character codes for multibyte representation range from 0 to 524287, but not all values in that range are valid. The values 128 through 255 are not entirely proper in multibyte text, but they can occur if you do explicit encoding and decoding (see Section 33.10.7 [Explicit Encoding], page 656). Some other character codes cannot occur at all in multibyte text. Only the ASCII codes 0 through 127 are completely legitimate in both representations.

**char-valid-p** *charcode* &**optional** *genericp*

[Function]

This returns `t` if *charcode* is valid (either for unibyte text or for multibyte text).

```
(char-valid-p 65)
  ⇒ t
(char-valid-p 256)
  ⇒ nil
(char-valid-p 2248)
  ⇒ t
```

If the optional argument `genericp` is non-`nil`, this function also returns `t` if `charcode` is a generic character (see Section 33.7 [Splitting Characters], page 645).

## 33.5 Character Sets

Emacs classifies characters into various *character sets*, each of which has a name which is a symbol. Each character belongs to one and only one character set.

In general, there is one character set for each distinct script. For example, `latin-iso8859-1` is one character set, `greek-iso8859-7` is another, and `ascii` is another. An Emacs character set can hold at most 9025 characters; therefore, in some cases, characters that would logically be grouped together are split into several character sets. For example, one set of Chinese characters, generally known as Big 5, is divided into two Emacs character sets, `chinese-big5-1` and `chinese-big5-2`.

ASCII characters are in character set `ascii`. The non-ASCII characters 128 through 159 are in character set `eight-bit-control`, and codes 160 through 255 are in character set `eight-bit-graphic`.

**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-list** [Function]  
 This function returns the value of `charset-list`. It is only provided for backward compatibility.

**char-charset** *character* [Function]  
 This function returns the name of the character set that *character* belongs to, or the symbol `unknown` if *character* is not a valid character.

**charset-plist** *charset* [Function]  
 This function returns the charset 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 are used for special purposes within Emacs.

**list-charset-chars** *charset* [Command]  
 This command displays a list of characters in the character set *charset*.

## 33.6 Characters and Bytes

In multibyte representation, each character occupies one or more bytes. Each character set has an *introduction sequence*, which is normally one or two bytes long. (Exception: the `ascii` character set and the `eight-bit-graphic` character set have a zero-length introduction sequence.) The introduction sequence is the beginning of the byte sequence for any character in the character set. The rest of the character's bytes distinguish it from the other characters in the same character set. Depending on the character set, there are either one or two distinguishing bytes; the number of such bytes is called the *dimension* of the character set.

**charset-dimension** *charset* [Function]

This function returns the dimension of *charset*; at present, the dimension is always 1 or 2.

**charset-bytes** *charset* [Function]

This function returns the number of bytes used to represent a character in character set *charset*.

This is the simplest way to determine the byte length of a character set's introduction sequence:

```
(- (charset-bytes charset)
  (charset-dimension charset))
```

## 33.7 Splitting Characters

The functions in this section convert between characters and the byte values used to represent them. For most purposes, there is no need to be concerned with the sequence of bytes used to represent a character, because Emacs translates automatically when necessary.

**split-char** *character* [Function]

Return a list containing the name of the character set of *character*, followed by one or two byte values (integers) which identify *character* within that character set. The number of byte values is the character set's dimension.

If *character* is invalid as a character code, `split-char` returns a list consisting of the symbol `unknown` and *character*.

```
(split-char 2248)
  ⇒ (latin-iso8859-1 72)
(split-char 65)
  ⇒ (ascii 65)
(split-char 128)
  ⇒ (eight-bit-control 128)
```

**make-char** *charset &optional code1 code2* [Function]

This function returns the character in character set *charset* whose position codes are *code1* and *code2*. This is roughly the inverse of `split-char`. Normally, you should specify either one or both of *code1* and *code2* according to the dimension of *charset*. For example,

```
(make-char 'latin-iso8859-1 72)
⇒ 2248
```

Actually, the eighth bit of both *code1* and *code2* is zeroed before they are used to index *charset*. Thus you may use, for instance, an ISO 8859 character code rather than subtracting 128, as is necessary to index the corresponding Emacs charset.

If you call `make-char` with no byte-values, the result is a *generic character* which stands for *charset*. A generic character is an integer, but it is *not* valid for insertion in the buffer as a character. It can be used in `char-table-range` to refer to the whole character set (see Section 6.6 [Char-Tables], page 93). `char-valid-p` returns `nil` for generic characters. For example:

```
(make-char 'latin-iso8859-1)
⇒ 2176
(char-valid-p 2176)
⇒ nil
(char-valid-p 2176 t)
⇒ t
(split-char 2176)
⇒ (latin-iso8859-1 0)
```

The character sets `ascii`, `eight-bit-control`, and `eight-bit-graphic` don't have corresponding generic characters. If *charset* is one of them and you don't supply *code1*, `make-char` returns the character code corresponding to the smallest code in *charset*.

### 33.8 Scanning for Character Sets

Sometimes it is useful to find out which character sets appear in a part of a buffer or a string. One use for this is in determining which coding systems (see Section 33.10 [Coding Systems], page 648) are capable of representing all of the text in question.

**charset-after &optional pos** [Function]

This function return the charset of a character in the current buffer at position *pos*. 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 that appear in the current buffer between positions *beg* and *end*.

The optional argument *translation* specifies a translation table to be used in scanning the text (see Section 33.9 [Translation of Characters], page 647). 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 the character sets that appear in the string *string*. It is just like `find-charset-region`, except that it applies to the contents of *string* instead of part of the current buffer.

### 33.9 Translation of Characters

A *translation table* is a char-table 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.

For instance, the coding-system `utf-8` has a translation table that maps characters of various charsets (e.g., `latin-iso8859-x`) into Unicode character sets. This way, it can encode Latin-2 characters into UTF-8. Meanwhile, `unify-8859-on-decoding-mode` operates by specifying `standard-translation-table-for-decode` to translate Latin-x characters into corresponding Unicode characters.

#### `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*.

You can also map one whole character set into another character set with the same dimension. To do this, you specify a generic character (which designates a character set) for *from* (see Section 33.7 [Splitting Characters], page 645). In this case, if *to* is also a generic character, its character set should have the same dimension as *from*'s. Then the translation table translates each character of *from*'s character set into the corresponding character of *to*'s character set. If *from* is a generic character and *to* is an ordinary character, then the translation table translates every character of *from*'s character set into *to*.

In decoding, the translation table's translations are applied to the characters that result from ordinary decoding. If a coding system has property `translation-table-for-decode`, that specifies the translation table to use. (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 33.10.1 [Basic Concepts of Coding Systems], page 648.) Otherwise, if `standard-translation-table-for-decode` is non-`nil`, decoding uses that table.

In 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 `translation-table-for-encode`, that specifies the translation table to use. Otherwise the variable `standard-translation-table-for-encode` specifies the translation table.

#### `standard-translation-table-for-decode`

[Variable]

This is the default translation table for decoding, for coding systems that don't specify any other translation table.

#### `standard-translation-table-for-encode`

[Variable]

This is the default translation table for encoding, for coding systems that don't specify any other translation table.

**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.

`set-buffer-file-coding-system` sets this variable so that your keyboard input gets translated into the character sets that the buffer is likely to contain. This variable automatically becomes buffer-local when set.

## 33.10 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.

### 33.10.1 Basic Concepts of Coding Systems

*Character code conversion* involves conversion between the encoding 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.

Most coding systems specify a particular character code for conversion, but some of them leave the choice unspecified—to be chosen heuristically for each file, based on the data.

In general, a coding system doesn't guarantee roundtrip identity: decoding a byte sequence using coding system, then encoding the resulting text in the same coding system, can produce a different byte sequence. However, the following coding systems do guarantee that the byte sequence will be the same as what you originally decoded:

```
chinese-big5 chinese-iso-8bit cyrillic-iso-8bit emacs-mule greek-iso-8bit hebrew-
iso-8bit iso-latin-1 iso-latin-2 iso-latin-3 iso-latin-4 iso-latin-5 iso-latin-8 iso-
latin-9 iso-safe japanese-iso-8bit japanese-shift-jis korean-iso-8bit raw-text
```

Encoding buffer text and then decoding the result can also fail to reproduce the original text. For instance, if you encode Latin-2 characters with `utf-8` and decode the result using the same coding system, you'll get Unicode characters (of charset `mule-unicode-0100-24ff`). If you encode Unicode characters with `iso-latin-2` and decode the result with the same coding system, you'll get Latin-2 characters.

*End of line conversion* handles three different conventions used on various systems for representing end of line in files. The Unix convention is to use the linefeed character (also called newline). The DOS convention is to use a carriage-return and a linefeed at the end of a line. The Mac convention is to use just carriage-return.

*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 that coding system to be a unibyte buffer. It 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` is equivalent to `raw-text-unix`: it specifies no conversion of either character codes or end-of-line.

The coding system `emacs-mule` specifies that the data is represented in the internal Emacs encoding. This is like `raw-text` in that no code conversion happens, but different in that the result is multibyte data.

`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.

### 33.10.2 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 for decoding 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 33.10.6 [Specifying Coding Systems], page 655), or implicitly using a default mechanism (see Section 33.10.5 [Default Coding Systems], page 653). But these methods may not completely specify what to do. For example, they may choose a coding system such as `undefined` 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 that was used to visit the current buffer. It is 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 33.10.4 [User-Chosen Coding Systems], page 652). 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 33.10.7 [Explicit Encoding], page 656) 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 29.18 [Window System Selections], page 550.

**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.

### 33.10.3 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`. Otherwise it signals an error with condition `coding-system-error`.

**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-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*.

**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 (see Section 33.10.7 [Explicit Encoding], page 656).

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.

**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.

See [Process Information], page 714, 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.

### 33.10.4 User-Chosen Coding Systems

**select-safe-coding-system** *from* *to* &optional *default-coding-system* [Function]  
*accept-default-p* *file*

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 *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 value of **default-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.

The optional argument *accept-default-p*, if non-*nil*, should be a function to determine whether a coding system selected without user interaction is acceptable. **select-safe-coding-system** calls this function with one argument, the base coding system of the selected coding system. If *accept-default-p* 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*, its value overrides the value of *accept-default-p*.

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.

Here are two functions you can use to let the user specify a coding system, with completion. See Section 20.6 [Completion], page 285.

#### `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 33.10 [Coding Systems], page 648.

### 33.10.5 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 33.10.6 [Specifying Coding Systems], page 655).

#### `auto-coding-regexp-alist`

[Variable]

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`

[Variable]

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.

**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**

[Variable]

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 undecoded 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`.

If a file has a ‘`coding:`’ tag, that takes precedence, so these functions won’t be called.

**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, 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.

### 33.10.6 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.
;; Assume crlf represents end-of-line.
(let ((coding-system-for-read 'emacs-mule-dos))
  (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.

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.

**inhibit-eol-conversion**

[Variable]

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 33.10.7 [Explicit Encoding], page 656).

### 33.10.7 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 characters whose codes are in the range 0 through 255. In a multibyte buffer or string, character codes 128 through 159 are represented by multibyte sequences, but this is invisible to Lisp programs.

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 25.3 [Reading from Files], page 440); 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 25.4 [Writing to Files], page 441), 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.

**encode-coding-region** *start end coding-system* [Command]

This command encodes the text from *start* to *end* according to coding system *coding-system*. The encoded text replaces the original text in the buffer. The result of encoding is logically a sequence of bytes, but the buffer remains multibyte if it was multibyte before.

This command returns the length of the encoded text.

**encode-coding-string** *string coding-system &optional nocopy* [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* [Command]

This command decodes the text from *start* to *end* according to coding system *coding-system*. The decoded text replaces the original text in the buffer. 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.

This command returns the length of the decoded text.

**decode-coding-string** *string coding-system &optional nocopy* [Function]

This function decodes the text in *string* according to coding system *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 sequence of byte values, but a multibyte string is acceptable.

**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.

### 33.10.8 Terminal I/O Encoding

Emacs can decode keyboard input using a coding system, 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** for encoding or decoding for the terminal.

**keyboard-coding-system** [Function]

This function returns the coding system that is in use for decoding keyboard input—or *nil* if no coding system is to be used.

**set-keyboard-coding-system** *coding-system* [Command]

This command specifies *coding-system* as the coding system to use for decoding keyboard input. If *coding-system* is *nil*, that means do not decode keyboard input.

**terminal-coding-system** [Function]

This function returns the coding system that is in use for encoding terminal output—or `nil` for no encoding.

**set-terminal-coding-system** *coding-system* [Command]

This command specifies *coding-system* as the coding system to use for encoding terminal output. If *coding-system* is `nil`, that means do not encode terminal output.

### 33.10.9 MS-DOS File Types

On MS-DOS and Microsoft Windows, Emacs guesses the appropriate end-of-line conversion for a file by looking at the file's name. This feature classifies files as *text files* and *binary files*. By “binary file” we mean a file of literal byte values that are not necessarily meant to be characters; Emacs does no end-of-line conversion and no character code conversion for them. On the other hand, the bytes in a text file are intended to represent characters; when you create a new file whose name implies that it is a text file, Emacs uses DOS end-of-line conversion.

**buffer-file-type** [Variable]

This variable, automatically buffer-local in each buffer, records the file type of the buffer's visited file. When a buffer does not specify a coding system with **buffer-file-coding-system**, this variable is used to determine which coding system to use when writing the contents of the buffer. It should be `nil` for text, `t` for binary. If it is `t`, the coding system is `no-conversion`. Otherwise, `undecided-dos` is used.

Normally this variable is set by visiting a file; it is set to `nil` if the file was visited without any actual conversion.

**file-name-buffer-file-type-alist** [User Option]

This variable holds an alist for recognizing text and binary files. Each element has the form `(regexp . type)`, where *regexp* is matched against the file name, and *type* may be `nil` for text, `t` for binary, or a function to call to compute which. If it is a function, then it is called with a single argument (the file name) and should return `t` or `nil`.

When running on MS-DOS or MS-Windows, Emacs checks this alist to decide which coding system to use when reading a file. For a text file, `undecided-dos` is used. For a binary file, `no-conversion` is used.

If no element in this alist matches a given file name, then **default-buffer-file-type** says how to treat the file.

**default-buffer-file-type** [User Option]

This variable says how to handle files for which **file-name-buffer-file-type-alist** says nothing about the type.

If this variable is non-`nil`, then these files are treated as binary: the coding system `no-conversion` is used. Otherwise, nothing special is done for them—the coding system is deduced solely from the file contents, in the usual Emacs fashion.

### 33.11 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 21.7.2 [Reading One Event], page 331, and Section 21.7.4 [Invoking the Input Method], page 334.

## 33.12 Locales

POSIX defines a concept of “locales” which 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 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`) for the default paper size measured in millimeters (locale items `PAPER_WIDTH` and `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.

## 34 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 30.2.7 [Skipping Characters], page 567. To search for changes in character properties, see Section 32.19.3 [Property Search], page 618.

### 34.1 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.

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 33.1 [Text Representations], page 640.

**search-forward** *string* &**optional** *limit noerror repeat* [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 upper bound to the search. (It must 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`. (It would be more consistent now to return the new position of point in that case, but some existing programs may depend on a value of `nil`.)

The argument *noerror* only affects valid searches which fail to find a match. Invalid arguments cause errors regardless of *noerror*.

If *repeat* is supplied (it must be a positive number), then the search is repeated that many times (each time starting at the end of the previous time's match). If these successive searches succeed, the function succeeds, moving point and returning its new value. Otherwise the search fails, with results depending on the value of *noerror*, as described above.

**search-backward** *string* &optional *limit* *noerror* *repeat* [Command]

This function searches backward from point for *string*. It is just like **search-forward** except that it searches backwards and leaves point at the beginning of the match.

**word-search-forward** *string* &optional *limit* *noerror* *repeat* [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.")
⇒ 35
```

```
----- 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 *repeat* is non-nil, then the search is repeated that many times. Point is positioned at the end of the last match.

**word-search-backward** *string* &optional *limit* *noerror* *repeat* [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.

## 34.2 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 11.10.1 [Intro to Buffer-Local], page 148.) Alternatively, you may change the value of `default-case-fold-search`, which is the default value of `case-fold-search` for buffers that do not override it.

Note that the user-level incremental search feature handles case distinctions differently. When given a lower case letter, it looks for a match of either case, but when given an upper case letter, it looks for an upper case letter only. But this has nothing to do with the searching functions used in Lisp code.

### `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 34.6.1 [Replacing Match], page 676.

### `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 they do ignore case.

### `default-case-fold-search`

[Variable]

The value of this variable is the default value for `case-fold-search` in buffers that do not override it. This is the same as `(default-value 'case-fold-search)`.

## 34.3 Regular Expressions

A *regular expression* (*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 convenient 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.

### 34.3.1 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.

### 34.3.1.1 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 that 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.

**Warning:** Nested repetition operators can run for an indefinitely long time, if they lead to ambiguous matching. For example, trying to match the regular expression ‘\((x+y\*)\)\*a’ against the string ‘xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxz’ could take hours before it ultimately fails. Emacs must try each way of grouping the ‘x’s before concluding that none of them can work. Even worse, ‘\((x\*)\)\*’ can match the null string in infinitely many ways, so it causes an infinite loop. To avoid these problems, check nested repetitions carefully, to make sure that they do not cause combinatorial explosions in backtracking.

‘+’

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.

‘\*?’ , ‘+?’ , ‘??’

These 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), from which it follows that ‘c[ad]\*r’ matches ‘cr’, ‘car’, ‘cdr’, ‘cadhaar’, 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.

Note that the usual regexp special characters are not special inside a character alternative. A completely different set of characters is special inside character alternatives: ‘]’, ‘-’ and ‘^’.

To include a ‘]’ in a character alternative, you must make it the first character. For example, ‘[]a]’ matches ‘]’ or ‘a’. To include a ‘-’, write ‘-’ as the first or last character of the character alternative, or put it after a range. Thus, ‘[]-]’ matches both ‘]’ and ‘-’.

To include ‘^’ in a character alternative, put it anywhere but at the beginning. The beginning and end of a range of multibyte characters must be in the same character set (see Section 33.5 [Character Sets], page 644). Thus, “[\x8e0-\x97c]” is invalid because character 0x8e0 (‘a’ with grave accent) is in the Emacs character set for Latin-1 but the character 0x97c (‘u’ with diaeresis) is in the Emacs character set for Latin-2. (We use Lisp string syntax to write that example, and a few others in the next few paragraphs, in order to include hex escape sequences in them.)

If a range starts with a unibyte character *c* and ends with a multibyte character *c*<sub>2</sub>, the range is divided into two parts: one is ‘*c*..?\\377’, the other is ‘*c*1..*c*2’, where *c*1 is the first character of the charset to which *c*<sub>2</sub> belongs.

You cannot always match all non-ASCII characters with the regular expression “[\\200-\\377]”. This works when searching a unibyte buffer or string (see Section 33.1 [Text Representations], page 640), but not in a multibyte buffer or

string, because many non-ASCII characters have codes above octal 0377. However, the regular expression "`[^\000-\177]`" does match all non-ASCII characters (see below regarding '`^`'), in both multibyte and unibyte representations, because only the ASCII characters are excluded.

A character alternative can also specify named character classes (see Section 34.3.1.2 [Char Classes], page 667). This is a POSIX feature whose syntax is '`[:class:]`'. 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.

**'`[^ ... ]`'** '`[^`' begins a *complemented character alternative*. This matches any character except the ones specified. Thus, '`[^a-zA-Z]`' matches all characters *except* 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`.

**'`^`'** 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 '`\C`', '`\(?:`' 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 2.3.8 [String Type], page 18), 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 ‘]’. So you should not quote these characters when they have no special meaning either. 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.

### 34.3.1.2 Character Classes

Here is a table of the classes you can use in a character alternative, and what they mean:

‘[:ascii:]’

This matches any ASCII character (codes 0–127).

‘[:alnum:]’

This matches any letter or digit. (At present, for multibyte characters, it matches anything that has word syntax.)

‘[:alpha:]’

This matches any letter. (At present, for multibyte characters, it matches anything that has word syntax.)

‘[:blank:]’

This matches space and tab only.

‘[:cntrl:]’

This matches any ASCII control character.

‘[:digit:]’

This matches ‘0’ through ‘9’. Thus, ‘[-+[:digit:]]’ matches any digit, as well as ‘+’ and ‘-’.

‘[:graph:]’

This matches graphic characters—everything except ASCII control characters, space, and the delete character.

`'[:lower:]'`

This matches any lower-case letter, as determined by the current case table (see Section 4.9 [Case Tables], page 60).

`'[:multibyte:]'`

This matches any multibyte character (see Section 33.1 [Text Representations], page 640).

`'[:nonascii:]'`

This matches any non-ASCII character.

`'[:print:]'`

This matches printing characters—everything except ASCII control characters and the delete character.

`'[:punct:]'`

This matches any punctuation character. (At present, for multibyte characters, it matches anything that has non-word syntax.)

`'[:space:]'`

This matches any character that has whitespace syntax (see Section 35.2.1 [Syntax Class Table], page 685).

`'[:unibyte:]'`

This matches any unibyte character (see Section 33.1 [Text Representations], page 640).

`'[:upper:]'`

This matches any upper-case letter, as determined by the current case table (see Section 4.9 [Case Tables], page 60).

`'[:word:]'`

This matches any character that has word syntax (see Section 35.2.1 [Syntax Class Table], page 685).

`'[:xdigit:]'`

This matches the hexadecimal digits: ‘0’ through ‘9’, ‘a’ through ‘f’ and ‘A’ through ‘F’.

### 34.3.1.3 Backslash Constructs in Regular Expressions

For the most part, ‘\’ followed by any character matches only that character. However, there are several exceptions: certain two-character sequences starting with ‘\’ that have special meanings. (The character after the ‘\’ in such a sequence is always ordinary when used on its own.) 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 34.5 [POSIX Regexp], page 676).

`\{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 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 ‘+’.

`\(\dots\)`

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 `\(\dots\)` 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.

`\(?:\dots\)`

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 any ordinary, non-shy groups.

`\digit`

matches the same text that matched the *digit*th occurrence of a grouping (`\(\dots\)`) 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 ‘\(\dots\)\’ 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 35 [Syntax Tables], page 684.

‘\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 35.2.1 [Syntax Class Table], page 685, for a list of syntax codes and the characters that stand for them.

‘\Scode’ matches any character whose syntax is not *code*.

‘\cc’ matches any character whose category is *c*. Here *c* is a character that represents a category: thus, ‘c’ for Chinese characters or ‘g’ for Greek characters in the standard category table.

‘\Cc’ matches any character whose category is not *c*.

The following regular expression constructs match the empty string—that is, they don’t use up 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 <i>not</i> 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 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.

### 34.3.2 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 34.8 [Standard Regexps], page 683.)

First, we show the regexp as a string in Lisp syntax to distinguish spaces from tab characters. 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 contrast, if you evaluate this string, you will see the following:

```
"[.?!] []\"')}] *\\($\\| $\\| \\t\\| \\)[ \\t\\n]*"  
⇒ "[.?!] []\"')}] *\\($\\| $\\| \\t\\| \\)[ \\t\\n]*"
```

In this 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\\|\\n\\|\\s\\)`

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.

### 34.3.3 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*.

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-"))
```

`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.

If the optional argument *paren* is non-*nil*, then the returned regular expression is always enclosed by at least one parentheses-grouping construct. If *paren* is `words`, then that construct is additionally surrounded by `'<'` and `'>'`.

This simplified definition of `regexp-opt` produces a regular expression which is equivalent to the actual value (but not as efficient):

```
(defun regexp-opt (strings paren)
  (let ((open-paren (if paren "\\\(" ""))
        (close-paren (if paren "\\\)" "")))
```

```
(concat open-paren
       (mapconcat 'regexp-quote strings "\\\\"")
       close-paren)))
```

**regexp-opt-depth** *regexp* [Function]

This function returns the total number of grouping constructs (parenthesized expressions) in *regexp*. (This does not include shy groups.)

## 34.4 Regular Expression Searching

In GNU Emacs, you can search for the next match for a regular expression 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 33.1 [Text Representations], page 640.

**re-search-forward** *regexp* &**optional** *limit noerror repeat* [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 *repeat* is supplied, it must be a positive number; the search is repeated that many times; each repetition starts at the end of the previous match. If all these successive searches succeed, the search succeeds, moving point and returning its new value. Otherwise the search fails. 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**.

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* *repeat* [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* [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.

After this function returns, the index of the first character beyond the match is available as (**match-end** 0). See Section 34.6 [Match Data], page 676.

```
(string-match
  "quick" "The quick brown fox jumped quickly." 8)
⇒ 27

(match-end 0)
⇒ 32
```

**looking-at** *regexp*

[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 updates the match data, which you can access using **match-beginning** and **match-end**. See Section 34.6 [Match Data], page 676.

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* &**optional limit**

[Function]

This function returns **t** if *regexp* matches text before point, 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 *limit*, which says not to search before *limit*. In this case, the match that is found must begin at or after *limit*.

```
----- Buffer: foo -----
I read "*The cat in the hat
comes back" twice.
----- Buffer: foo -----  

(looking-back "read \"\" 3)
⇒ t
(looking-back "read \"\" 4)
⇒ nil
```

**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.

## 34.5 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. This is because POSIX backtracking conflicts with the semantics of non-greedy repetition.

**posix-search-forward** *regexp* &optional *limit noerror repeat* [Function]

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 repeat* [Function]

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* [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* [Function]

This is like **string-match** except that it performs the full backtracking specified by the POSIX standard for regular expression matching.

## 34.6 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.

### 34.6.1 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 replaces the text in the buffer (or in *string*) that was matched by the last search. It replaces that text with *replacement*.

If you did the last search in a buffer, you should specify `nil` for *string* and make sure that the current buffer when you call `replace-match` is the one in which you did the searching or matching. Then `replace-match` does the replacement by editing the buffer; it leaves point at the end of the replacement text, and returns `t`.

If you did the search in a string, pass the same string as *string*. Then `replace-match` does the replacement by constructing and returning a new string.

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:

- ‘\&’        ‘\&’ stands for the entire text being replaced.
- ‘\n’        ‘\n’, where *n* is a digit, 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.
- ‘\\’        ‘\\’ stands for a single ‘\’ in the replacement text.

These substitutions occur after case conversion, if any, so 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 ‘\(\ba\*\r\)''.

### 34.6.2 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 34.6.4 [Saving Match Data], page 680) around the call to functions that could perform another search.

A search which fails may or may not alter the match data. In the past, a failing search did not do this, but we may change it in the future. So 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.

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]

This function returns the position of the start of text matched by the last regular expression searched for, or 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))
⇒ (9 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.)

### 34.6.3 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 what text 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`.

#### 34.6.4 Saving and Restoring the Match Data

When you call a function that may do a 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) ; Perhaps 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 37.9.2 [Filter Functions], page 718) and process sentinels (see Section 37.10 [Sentinels], page 721).

## 34.7 Search and Replace

If you want to find all matches for a regexp in part of the buffer, and replace them, the best 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 34.6.1 [Replacing the Text that Matched], page 676, for a description of `replace-match`.

However, replacing matches in a string is more complex, especially if you want to do it efficiently. So Emacs provides a function to do this.

**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*, so matches starting before that index are not changed.

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*.

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* [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 *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 the case of them.

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**.

#### **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.”

#### **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.

**edit** Enter a recursive edit to deal with this question—instead of any other action that would normally be taken.

**delete-and-edit**

Delete the text being considered, then enter a recursive edit to replace it.

**recenter** Redisplay and center the window, then ask the same question again.

**quit** Perform a quit right away. Only **y-or-n-p** and related functions use this answer.

**help** Display some help, then ask again.

## 34.8 Standard Regular Expressions Used in Editing

This section describes some variables that hold regular expressions used for certain purposes in editing:

**page-delimiter**

[Variable]

This is the regular expression describing line-beginnings that separate pages. The default value is "`^\014`" (i.e., "`^^L`" or "`^\C-1`"); 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**

[Variable]

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**

[Variable]

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**

[Variable]

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**, the default, then the function **sentence-end** has to construct 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 601), **sentence-end-without-period** and **sentence-end-without-space**.

## 35 Syntax Tables

A *syntax table* specifies the syntactic textual function of each character. This information is used by the *parsing functions*, the complex movement commands, and others to determine where words, symbols, and other syntactic constructs begin and end. The current syntax table controls the meaning of the word motion functions (see Section 30.2.2 [Word Motion], page 561) and the list motion functions (see Section 30.2.6 [List Motion], page 566), as well as the functions in this chapter.

### 35.1 Syntax Table Concepts

A syntax table is a char-table (see Section 6.6 [Char-Tables], page 93). The element at index *c* describes the character with code *c*. The element's value should be a list that encodes the syntax of the character in question.

Syntax tables are used only for moving across text, not for the Emacs Lisp reader. Emacs Lisp uses built-in syntactic rules when reading Lisp expressions, and these rules 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.)

Each buffer has its own major mode, and each major mode has its own idea of the syntactic 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, Emacs makes the choice of syntax table local to each buffer. Typically, each major mode has its own syntax table and installs that table in each buffer that uses that mode. Changing this table alters the syntax in all those buffers as well as in any buffers subsequently put in that mode. Occasionally several similar modes share one syntax table. See Section 23.2.8 [Example Major Modes], page 394, for an example of how to set up a syntax table.

A syntax table can inherit the data for some characters from the standard syntax table, while specifying other characters itself. The “inherit” syntax class means “inherit this character’s syntax from the standard syntax table.” Just changing the standard syntax for a character affects all syntax tables that inherit from it.

**`syntax-table-p object`** [Function]

This function returns `t` if *object* is a syntax table.

### 35.2 Syntax Descriptors

This section describes the syntax classes and flags that denote the syntax of a character, and how they are represented as a *syntax descriptor*, which is a Lisp string that you pass to `modify-syntax-entry` to specify the syntax you want.

The syntax table specifies a syntax class for 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 class is designated by a mnemonic character, which serves as the name of the class when you need to specify a class. Usually the 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 gives “escape character” syntax, regardless of what syntax ‘\’ currently has.

A syntax descriptor is a Lisp string that specifies a syntax class, a matching character (used only for the parenthesis classes) and flags. The first character is the designator for a syntax class. The second character is the character to match; if it is unused, put a space there. Then come the characters for any desired flags. If no matching character or flags are needed, one character 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).

### 35.2.1 Table of Syntax Classes

Here is a table of syntax classes, the characters that stand for them, their meanings, and examples of their use.

**whitespace character** [Syntax class]

*Whitespace characters* (designated by ‘ ’ or ‘-’) 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, newline and formfeed are classified as whitespace in almost all major modes.

**word constituent** [Syntax class]

*Word constituents* (designated by ‘w’) are parts of words in human languages, and are typically used in variable and command names in programs. All upper- and lower-case letters, and the digits, are typically word constituents.

**symbol constituent** [Syntax class]

*Symbol constituents* (designated by ‘\_’) are the extra characters that are used in variable and command names along with word constituents. For example, the symbol constituents class is used in Lisp mode to indicate that certain characters may be part of symbol names even though they are not part of English words. These characters are ‘\$&\*+-<>’. In standard C, the only non-word-constituent character that is valid in symbols is underscore (‘\_’).

**punctuation character** [Syntax class]

*Punctuation characters* (designated by ‘.’) are those characters that are used as punctuation in English, or are used in some way 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 character** [Syntax class]

**close parenthesis character** [Syntax class]

Open and close *parenthesis characters* are 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 38.19 [Blinking], page 805.

The class of open parentheses is designated by ‘(’, and that of close parentheses by ‘)’.

In English text, 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 quote**

[Syntax class]

*String quote characters* (designated by “”) are used in many languages, including Lisp and C, 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 single-quote (‘’’) for character constants.

English text has no string quote characters because English is not a programming language. Although quotation marks are used in English, we do not want them to turn off the usual syntactic properties of other characters in the quotation.

#### **escape-syntax character**

[Syntax class]

An *escape character* (designated by ‘\’) starts an escape sequence such as is used in C 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 30.2.2 [Word Motion], page 561.

#### **character quote**

[Syntax class]

A *character quote character* (designated by ‘/’) quotes 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 30.2.2 [Word Motion], page 561.

This class is used for backslash in TeX mode.

#### **paired delimiter**

[Syntax class]

*Paired delimiter characters* (designated by ‘\$’) are like 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 prefix**

[Syntax class]

An *expression prefix operator* (designated by ‘’’) is 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 starter** [Syntax class]  
**comment ender** [Syntax class]

The *comment starter* and *comment ender* characters are used in various languages to delimit comments. These classes are designated by ‘<’ and ‘>’, respectively.

English text has no comment characters. In Lisp, the semicolon (‘;’) starts a comment and a newline or formfeed ends one.

**inherit standard syntax** [Syntax class]

This syntax class does not specify a particular syntax. It says to look in the standard syntax table to find the syntax of this character. The designator for this syntax class is ‘@’.

**generic comment delimiter** [Syntax class]

A *generic comment delimiter* (designated by ‘!’) starts or ends 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 35.4 [Syntax Properties], page 690). 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 delimiter** [Syntax class]

A *generic string delimiter* (designated by ‘|’) starts or ends 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 35.4 [Syntax Properties], page 690). 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.

### 35.2.2 Syntax Flags

In addition to the classes, entries for characters in a syntax table can specify flags. There are seven possible flags, represented by the characters ‘1’, ‘2’, ‘3’, ‘4’, ‘b’, ‘n’, and ‘p’.

All the flags except ‘n’ and ‘p’ are used to describe multi-character comment delimiters. The digit flags 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 (‘\*/’).

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. Emacs supports two comment styles simultaneously in any one syntax table. This is for the sake of C++. Each style of comment syntax has its own comment-start sequence and its own comment-end sequence. Each comment must stick to one style or the other; thus, if it starts with the comment-start sequence of style “b,” it must also end with the comment-end sequence of style “b.”

The two comment-start sequences must begin with the same character; only the second character may differ. Mark the second character of the “b”-style comment-start sequence with the ‘b’ flag.

A comment-end sequence (one or two characters) applies to the “b” style if its first character has the ‘b’ flag set; otherwise, it applies to the “a” style.

The appropriate comment syntax settings for C++ are as follows:

‘/’	‘124b’
‘*’	‘23’
newline	‘>b’

This defines four comment-delimiting sequences:

‘/*’	This is a comment-start sequence for “a” style because the second character, ‘*’, does not have the ‘b’ flag.
‘//’	This is a comment-start sequence for “b” style because the second character, ‘/’, does have the ‘b’ flag.
‘*/’	This is a comment-end sequence for “a” style because the first character, ‘*’, does not have the ‘b’ flag.
newline	This is a comment-end sequence for “b” style, because the newline character has the ‘b’ flag.

- ‘n’ on a comment delimiter character specifies that this kind of comment can be nested. For a two-character comment delimiter, ‘n’ on either character makes it nestable.
- ‘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 35.5 [Motion and Syntax], page 691.

### 35.3 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, with all values initialized to `nil`. If *table* is non-`nil`, it becomes the parent of the new syntax table, otherwise the standard syntax table is the parent. Like all char-tables, a syntax table inherits from its parent. Thus the original syntax of all characters in the returned syntax table is determined by the parent. See Section 6.6 [Char-Tables], page 93.

Most major mode syntax tables are created in this way.

**copy-syntax-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 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*. 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* specifies the desired syntax; this is a string beginning with a class designator character, and optionally containing a matching character and flags as well. See Section 35.2 [Syntax Descriptors], page 684.

This function always returns `nil`. The old syntax information in the table for this character is discarded.

An error is signaled if the first character of the syntax descriptor is not one of the seventeen syntax class designator characters. An error is also signaled if *char* is not a character.

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 mnemonic designator character. This returns *only* the class, not any matching parenthesis or flags.

An error is signaled if *char* is not a character.

The following examples apply to C mode. The first example shows that the syntax class of space is whitespace (represented by a space). The second example shows that

the syntax of ‘/’ is punctuation. This does not show the fact that it is also part of comment-start and -end sequences. The third example shows that open parenthesis is in the class of open parentheses. This does not show the fact that it has a matching character, ‘)’.

```
(string (char-syntax ?\s))
⇒ " "
(string (char-syntax ?/))
⇒ "."
(string (char-syntax ?\()))
⇒ "("
```

We use `string` to make it easier to see the character returned by `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.

`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.

## 35.4 Syntax Properties

When the syntax table is not flexible enough to specify the syntax of a language, you can use `syntax-table` text properties to override the syntax table for specific character occurrences in the buffer. See Section 32.19 [Text Properties], page 615. You can use Font Lock mode to set `syntax-table` text properties. See Section 23.6.9 [Setting Syntax Properties], page 421.

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 this occurrence of the character.

`(syntax-code . matching-char)`

A cons cell of this format specifies the syntax for this occurrence of the character. (see Section 35.8 [Syntax Table Internals], page 695)

`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 pay attention to syntax text properties. Otherwise they use only the current syntax table.

## 35.5 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.

## 35.6 Parsing Expressions

This section describes functions for parsing and scanning balanced expressions, also known as *sexps*. Basically, a *sexp* is either a balanced parenthetical grouping, a string, or a symbol name (a sequence of characters whose syntax is either word constituent or symbol constituent). However, characters whose syntax is expression prefix 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 30.2.6 [List Motion], page 566, 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.

### 35.6.1 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, parenthesis depth counting begins from that value. The only candidates for stopping are places where the depth in parentheses becomes zero; **scan-lists** counts *count* such places and then stops. Thus, a positive value for *depth* means go out *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 buffer (or its accessible portion), and the depth is not zero, an error is signaled. If the depth is zero but the count is not used up, **nil** is returned.

**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.

### 35.6.2 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 (see next section) that the parser would reach at position *pos* starting from the beginning of the buffer. This is equivalent to (`parse-partial-sexp (point-min) pos`), except that `syntax-ppss` uses a cache to speed up the computation. Due to this optimization, the 2nd value (previous complete subexpression) and 6th value (minimum parenthesis depth) of the returned parser state are not meaningful.

`syntax-ppss` automatically hooks itself to `before-change-functions` to keep its cache consistent. But updating can fail if `syntax-ppss` is called while `before-change-functions` is temporarily let-bound, or if the buffer is modified without obeying the hook, such as when using `inhibit-modification-hooks`. For this reason, it is sometimes necessary to flush the cache manually.

**syntax-ppss-flush-cache beg** [Function]

This function flushes the cache used by `syntax-ppss`, starting at position *beg*.

Major modes can make `syntax-ppss` run faster by specifying where it needs to start parsing.

**syntax-begin-function** [Variable]

If this is non-`nil`, it should be a function that moves to an earlier buffer position where the parser state is equivalent to `nil`—in other words, a position outside of any comment, string, or parenthesis. `syntax-ppss` uses it to further optimize its computations, when the cache gives no help.

### 35.6.3 Parser State

A *parser state* is a list of ten elements describing the final state of parsing text syntactically as part of an expression. The parsing functions in the following sections return a parser state as the value, and in some cases accept one as an argument also, so that you can resume parsing after it stops. 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 start of the defun and 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 comment (of either style), or the comment nesting level if inside a kind of comment that can be nested.
5. `t` if point is just after a quote character.
6. The minimum parenthesis depth encountered during this scan.
7. What kind of comment is active: `nil` for a comment of style “a” or when not inside a comment, `t` for a comment of style “b,” 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. Internal data for continuing the parsing. The meaning of this data is subject to change; it is used if you pass this list as the `state` argument to another call.

Elements 1, 2, and 6 are ignored in a state which you pass as an argument to continue parsing, and elements 8 and 9 are used only in trivial cases. Those elements serve primarily to convey information to the Lisp program which does the parsing.

One additional piece of useful information is available from a parser state using this function:

`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.

We have provided this access function rather than document how the data is represented in the state, because we plan to change the representation in the future.

### 35.6.4 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 when it comes to the start of a comment. If `stop-comment` is the symbol `syntax-table`, parsing stops after the start of a comment or a string, or the end of a 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.

### 35.6.5 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, 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 35.4 [Syntax Properties], page 690).

You can use `forward-comment` to move forward or backward over one comment or several comments.

## 35.7 Some Standard Syntax Tables

Most of the major modes in Emacs have their own syntax tables. Here are several of them:

**standard-syntax-table**

[Function]

This function returns the standard syntax table, which is the syntax table used in Fundamental mode.

**text-mode-syntax-table**

[Variable]

The value of this variable is the syntax table used in Text mode.

**c-mode-syntax-table**

[Variable]

The value of this variable is the syntax table for C-mode buffers.

**emacs-lisp-mode-syntax-table**

[Variable]

The value of this variable is the syntax table used in Emacs Lisp mode by editing commands. (It has no effect on the Lisp `read` function.)

## 35.8 Syntax Table Internals

Lisp programs don't usually work with the elements directly; the Lisp-level syntax table functions usually work with syntax descriptors (see Section 35.2 [Syntax Descriptors], page 684). Nonetheless, here we document the internal format. This format is used mostly when manipulating syntax properties.

Each element of a syntax table is a cons cell of the form `(syntax-code . matching-char)`. The CAR, `syntax-code`, is an integer that encodes the syntax class, and any flags. The CDR, `matching-char`, is non-*nil* if a character to match was specified.

This table gives the value of `syntax-code` which corresponds to each syntactic type.

<i>Integer Class</i>	<i>Integer Class</i>	<i>Integer Class</i>
0 whitespace	5 close parenthesis	10 character quote
1 punctuation	6 expression prefix	11 comment-start
2 word	7 string quote	12 comment-end
3 symbol	8 paired delimiter	13 inherit

4 open parenthesis	9 escape	14 generic comment
15 generic string		

For example, the usual syntax value for ‘(’ is (4 . 41). (41 is the character code for ‘)’.)

The 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	Prefix Flag
‘1’ (lsh 1 16)	‘4’ (lsh 1 19)	‘b’ (lsh 1 21)
‘2’ (lsh 1 17)	‘p’ (lsh 1 20)	‘n’ (lsh 1 22)
‘3’ (lsh 1 18)		

**string-to-syntax desc** [Function]

This function returns the internal form corresponding to the syntax descriptor *desc*, a cons cell (*syntax-code* . *matching-char*).

**syntax-after pos** [Function]

This function returns the syntax code of 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 30.4 [Narrowing], page 569), this function returns **nil**.

**syntax-class syntax** [Function]

This function returns the syntax class of the syntax code *syntax*. (It masks off the high 16 bits that hold the flags encoded in the syntax descriptor.) If *syntax* is **nil**, it returns **nil**; this is so evaluating the expression

```
(syntax-class (syntax-after pos))
```

where *pos* is outside the buffer’s accessible portion, will yield **nil** without throwing errors or producing wrong syntax class codes.

## 35.9 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 93). 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*.

**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\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** *character category &optional table reset* [Function]

This function modifies the category set of *character* in category table *table* (which defaults to the current buffer's category table).

Normally, it modifies the 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.

## 36 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 containing a symbol for each abbreviation. The symbol's name is the abbreviation; its value is the expansion; its function definition is the hook function to do the expansion (see Section 36.3 [Defining Abbrevs], page 700); its property list cell typically contains the use count, the number of times the abbreviation has been expanded. Alternatively, the use count is on the `count` property and the system-abbrev flag is on the `system-type` property. Abbrevs with a non-`nil` `system-type` property are called “system” abbrevs. They are usually defined by modes or packages, instead of by the user, and are treated specially in certain respects.

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 an extremely nonstandard way. See Section 8.3 [Creating Symbols], page 104.

For the user-level commands for abbrevs, see section “Abbrev Mode” in *The GNU Emacs Manual*.

### 36.1 Setting Up Abbrev Mode

Abbrev mode is a minor mode controlled by the value of the variable `abbrev-mode`.

`abbrev-mode` [Variable]

A non-`nil` value of this variable turns on the automatic expansion of abbrevs when their abbreviations are inserted into a buffer. If the value is `nil`, abbrevs may be defined, but they are not expanded automatically.

This variable automatically becomes buffer-local when set in any fashion.

`default-abbrev-mode` [Variable]

This is the value of `abbrev-mode` for buffers that do not override it. This is the same as `(default-value 'abbrev-mode)`.

### 36.2 Abbrev Tables

This section describes how to create and manipulate abbrev tables.

`make-abbrev-table` [Function]

This function creates and returns a new, empty abbrev table—an obarray containing no symbols. It is a vector filled with zeros.

`clear-abbrev-table table` [Function]

This function undefines all the abbrevs in abbrev table `table`, leaving it empty. It always returns `nil`.

**copy-abbrev-table** *table*

[Function]

This function returns a copy of abbrev table *table*—a new abbrev table that contains the same abbrev definitions. The only difference between *table* and the returned copy is that this function sets the property lists of all copied abbrevs to 0.

**define-abbrev-table** *tabname* *definitions*

[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* *usecount* *system-flag*). If an element of *definitions* has length less than five, omitted elements default to *nil*. A value of *nil* for *usecount* is equivalent to zero. The return value is always *nil*.

If this function is called more than once for the same *tabname*, subsequent calls add the definitions in *definitions* to *tabname*, rather than overriding 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. The return value is always *nil*.

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.)

### 36.3 Defining Abbrevs

**define-abbrev** is the low-level basic function for defining an abbrev in a specified abbrev table. When major modes predefine standard abbrevs, they should call **define-abbrev** and specify *t* for *system-flag*. Be aware that any saved non-“system” abbrevs are restored at startup, i.e. before some major modes are loaded. Major modes should therefore not assume that when they are first loaded their abbrev tables are empty.

**define-abbrev** *table* *name* *expansion* &**optional** *hook* *count* *system-flag*

[Function]

This function defines an abbrev named *name*, in *table*, to expand to *expansion* and call *hook*. The return value is *name*.

The value of *count*, if specified, initializes the abbrev’s usage-count. If *count* is not specified or *nil*, the use count is initialized to zero.

The argument *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` also returns `nil`, as if expansion had not really occurred.

If `system-flag` is non-`nil`, that marks the abbrev as a “system” abbrev with the `system-type` property. Unless `system-flag` has the value `force`, a “system” abbrev will not overwrite an existing definition for a non-“system” abbrev of the same name.

Normally the function `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 won’t be 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.

## 36.4 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 15.1 [How Programs Do Loading], page 201). However, the function `quietly-read-abbrev-file` is provided as a more convenient interface.

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.

#### `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. `save-abbrevs` is set to `t` so that changes will be saved.

This function does not display any messages. It returns `nil`.

#### `save-abbrevs`

[User Option]

A non-`nil` value for `save-abbrevs` means that Emacs should offer the user to save abbrevs 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.

**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. If *filename* is *nil* or omitted, *abbrev-file-name* is used. This function returns *nil*.

## 36.5 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*. The value returned is *nil* if that abbrev is not defined. The optional second argument *table* is the abbrev table to look it up in. 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). If *abbrev* is not a valid abbrev, the function returns *nil*. 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. The command returns the abbrev symbol if it did expansion, *nil* otherwise.

If the abbrev symbol has a hook function which is a symbol whose *no-self-insert* property is non-*nil*, and if the hook function returns *nil* as its value, then *expand-abbrev* returns *nil* even though expansion did occur.

**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.

**pre-abbrev-expand-hook**

[Variable]

This is a normal hook whose functions are executed, in sequence, just before any expansion of an abbrev. See Section 23.1 [Hooks], page 382. Since it is a normal hook, the hook functions receive no arguments. However, they can find the abbrev to be expanded by looking in the buffer before point. Running the hook is the first thing that `expand-abbrev` does, and so a hook function can be used to change the current abbrev table before abbrev lookup happens. (Although you have to do this carefully. See the example below.)

The following sample code shows a simple use of `pre-abbrev-expand-hook`. 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. Then you can put the following code in your ‘`.emacs`’ file. See Section 36.6 [Standard Abbrev Tables], page 704, for the definitions of `local-abbrev-table` and `text-mode-abbrev-table`.

```
(defun foo-mode-pre-abbrev-expand ()
  (when (save-excursion (forward-line 0) (eq (char-after) ?#))
    (let ((local-abbrev-table text-mode-abbrev-table)
          ;; Avoid infinite loop.
          (pre-abbrev-expand-hook nil))
      (expand-abbrev))
    ;; We have already called 'expand-abbrev' in this hook.
    ;; Hence we want the "actual" call following this hook to be a no-op.)
```

```
(setq abbrev-start-location (point-max)
      abbrev-start-location-buffer (current-buffer)))

(add-hook 'foo-mode-hook
 #'(lambda ()
     (add-hook 'pre-abbrev-expand-hook
 'foo-mode-pre-abbrev-expand
 nil t)))
```

Note that `foo-mode-pre-abbrev-expand` just returns `nil` without doing anything for lines not starting with '#'. Hence abbrevs expand normally using `foo-mode-abbrev-table` as local abbrev table for such lines.

## 36.6 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.

**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 and Emacs Lisp mode.

## 37 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.

**processp** *object* [Function]

This function returns `t` if *object* is a process, `nil` otherwise.

### 37.1 Functions that Create Subprocesses

There are three functions that create a new subprocess in which to run a program. One of them, `start-process`, creates an asynchronous process and returns a process object (see Section 37.4 [Asynchronous Processes], page 710). The other two, `call-process` and `call-process-region`, create a synchronous process and do not return a process object (see Section 37.3 [Synchronous Processes], page 707).

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 function’s *program* argument specifies 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 25.8.4 [File Name Expansion], page 457). `nil` in this list refers to `default-directory`.

Executing a program can also try adding suffixes to the specified name:

**exec-suffixes** [Variable]

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; it may not contain any command-line arguments. You must use *args* to provide those.

Each of the subprocess-creating functions has a *buffer-or-name* argument which specifies where the standard 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 filter function handles it. (See Section 37.9.2 [Filter Functions], page 718, and Chapter 19 [Read and Print], page 268.) Normally, you should avoid having multiple processes send output to the same buffer because their output would be intermixed randomly.

All three of the subprocess-creating functions have a `&rest` argument, `args`. The `args` must all be strings, and they are supplied to `program` as separate command line arguments. 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 gets its current directory from the value of `default-directory` (see Section 25.8.4 [File Name Expansion], page 457).

The subprocess inherits its environment from Emacs, but you can specify overrides for it with `process-environment`. See Section 39.3 [System Environment], page 819.

#### `exec-directory`

[Variable]

The value of this variable is a string, the name of a directory that contains programs that come with GNU Emacs, programs 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`).

The value of `exec-path` is used by `call-process` and `start-process` when the `program` argument is not an absolute file name.

## 37.2 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`

[Function]

This function returns a string which 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.

`; ; 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 -c "
       (shell-quote-argument oldfile)
       " "
       (shell-quote-argument newfile))
```

### 37.3 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<sup>1</sup> 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 21.10 [Quitting], page 338.

The synchronous subprocess functions return an indication of how the process terminated.

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 33.10 [Coding Systems], page 648.

**`call-process`** *program* &**`optional`** *infile* *destination* *display* &**`rest`** *args* [Function]

This function calls *program* in a separate process and waits for it to finish.

The standard input for the 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 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.

---

<sup>1</sup> On other systems, Emacs uses a Lisp emulation of `ls`; see Section 25.9 [Contents of Directories], page 462.

```
(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.

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 37.9 [Output from Processes], page 717.)

Otherwise the function `call-process` does no redisplay, and the results become visible on the screen only when Emacs redisperses that buffer in the normal course of events.

The remaining arguments, *args*, are strings that specify command line arguments for the program.

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.

In the examples below, the buffer '`foo`' is current.

```
(call-process "pwd" nil t)
⇒ 0

----- Buffer: foo -----
/usr/user/lewis/manual
----- Buffer: foo -----


(call-process "grep" nil "bar" nil "lewis" "/etc/passwd")
⇒ 0

----- Buffer: bar -----
lewis:5LTsHm66CSWKg:398:21:Bil Lewis:/user/lewis:/bin/csh

----- Buffer: bar -----
```

Here is a good example of the use of `call-process`, which used to be found in the definition of `insert-directory`:

```
(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 handler based on the value of the variable `default-directory`. The current working directory of the subprocess is `default-directory`.

The arguments are handled in almost the same way as for `call-process`, with the following differences:

Some file handlers may not support all combinations and forms of the arguments `infile`, `buffer`, and `display`. For example, some file handlers might behave as if `display` were `nil`, regardless of the value actually passed. As another example, some file handlers might not support separating standard output and error output by way of the `buffer` argument.

If a file handler is invoked, it determines the program to run based on the first argument `program`. For instance, consider that a handler for remote files is invoked. Then the path that is used for searching the program might be different than `exec-path`.

The second argument `infile` may invoke a file handler. The file handler could be different from the handler chosen for the `process-file` function itself. (For example, `default-directory` could be on a remote host, whereas `infile` is on another 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.

**call-process-region** start end program &optional delete destination [Function]  
display &rest args

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).

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 -----
```

The `shell-command-on-region` command uses `call-process-region` like this:

```
(call-process-region
  start end
  shell-file-name      ; Name of program.
  nil                 ; Do not delete region.
  buffer              ; Send output to buffer.
  nil                 ; No redisplay during output.
  "-c" command)       ; Arguments for the shell.
```

**call-process-shell-command** *command* &optional *infile destination* [Function]  
*display &rest args*

This function executes the shell command *command* synchronously in a separate process. The final arguments *args* are additional arguments to add at the end of *command*. The other arguments are handled as in `call-process`.

**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.

## 37.4 Creating an Asynchronous Process

After an asynchronous process is created, Emacs and the subprocess both continue running immediately. The process thereafter runs in parallel with Emacs, and the two can communicate with each other using the functions described in the following sections. However, communication is only partially asynchronous: Emacs sends data to the process only when certain functions are called, and Emacs accepts data from the process only when Emacs is waiting for input or for a time delay.

Here we describe how to create an asynchronous process.

**start-process** *name buffer-or-name program &rest args* [Function]  
This function creates a new asynchronous subprocess and starts the program *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; if a process with this name already exists, then *name* is modified (by appending '*<1>*', etc.) to be unique. The buffer *buffer-or-name* is the buffer to associate with the process.

The remaining arguments, *args*, are strings that specify command line arguments for the program.

In the example below, the first process is started and runs (rather, sleeps) for 100 seconds. 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" "/user/lewis/bin")
⇒ #<process my-process<1>>

----- Buffer: foo -----
total 2
lrwxrwxrwx 1 lewis    14 Jul 22 10:12 gnuemacs --> /emacs
-rwxrwxrwx 1 lewis    19 Jul 30 21:02 lemon

Process my-process<1> finished

Process my-process finished
----- Buffer: foo -----
```

**start-process-shell-command** *name buffer-or-name command &rest command-args* [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 name, and *command-args* are the arguments for the shell command. The variable **shell-file-name** specifies which shell to use.

The point of running a program through the shell, rather than directly with **start-process**, is so that you can employ shell features such as wildcards in the arguments. It follows that if you include an arbitrary user-specified arguments in the command, you should quote it with **shell-quote-argument** first, so that any special shell characters do *not* have their special shell meanings. See Section 37.2 [Shell Arguments], page 706.

**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.

PTYs are usually preferable for processes visible to the user, as in Shell mode, because they allow job control (*C-c*, *C-z*, etc.) to work between the process and its children, whereas pipes do not. For subprocesses used for internal purposes by programs, it is often better to use a pipe, because they are more efficient. In addition, the total number of PTYs is limited on many systems and it is good not to waste them.

The value of **process-connection-type** takes effect when **start-process** is called. So you can specify how to communicate with one subprocess by binding the variable around the call to **start-process**.

```
(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 37.6 [Process Information], page 712).

## 37.5 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 delete a terminated process explicitly 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 if it has one. See Section 37.10 [Sentinels], page 721.

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 process`

[Function]

This function deletes a process, killing it with a `SIGKILL` signal. 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.) Calling `delete-process` on a running process terminates it, updates the process status, and runs the sentinel (if any) 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).

```
(delete-process "*shell*")
⇒ nil
```

## 37.6 Process Information

Several functions return information about processes. `list-processes` is provided for interactive use.

### `list-processes &optional query-only`

[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`.

If `query-only` is non-`nil` then it lists only processes whose query flag is non-`nil`. See Section 37.11 [Query Before Exit], page 723.

### `process-list`

[Function]

This function returns a list of all processes that have not been deleted.

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

### `get-process name`

[Function]

This function returns the process named `name`, or `nil` if there is none. An error is signaled if `name` is not a string.

```
(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.

```
(process-command (get-process "shell"))
⇒ ("/bin/csh" "-i")
```

**process-id** *process*

[Function]

This function returns the PID of *process*. This is an integer 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.

**process-name** *process*

[Function]

This function returns the name of *process*.

**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, a process name (string) or a buffer name (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 connection that is open.
- closed** for a network 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 "shell")
⇒ run
(process-status (get-buffer "*shell*"))
⇒ run
x
⇒ #<process xx<1>>
(process-status x)
⇒ exit
```

For a network connection, **process-status** returns one of the symbols **open** or **closed**. The latter means that the other side closed the connection, or Emacs did **delete-process**.

**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.

**process-tty-name** *process* [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 terminal (see **process-connection-type** in Section 37.4 [Asynchronous Processes], page 710).

**process-coding-system** *process* [Function]

This function returns a cons cell describing the coding systems in use for decoding output from *process* and for encoding input to *process* (see Section 33.10 [Coding Systems], page 648). The value has this form:

*(coding-system-for-decoding . coding-system-for-encoding)*

**set-process-coding-system** *process* &**optional** *decoding-system* [Function]  
*encoding-system*

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*.

## 37.7 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. The data appears on the “standard input” of the subprocess.

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 37.6 [Process Information], page 712). Otherwise, the coding system comes from **coding-system-for-write**, if that is non-`nil`; or else from the defaulting mechanism (see Section 33.10.5 [Default Coding Systems], page 653).

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, 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. If it is `nil`, the current buffer's process is used.

The function returns `nil`.

```
(process-send-string "shell<1>" "ls\n")
⇒ nil
```

```
----- Buffer: *shell* -----
...
introduction.texi      syntax-tables.texi~
introduction.texi~    text.texi
introduction.txt      text.texi~
...
----- Buffer: *shell* -----
```

#### `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` *process*

[Function]

This function will tell you whether a subprocess has given control of its terminal to its own child process. The value is `t` if this is true, or if Emacs cannot tell; it is `nil` if Emacs can be certain that this is not so.

## 37.8 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.)

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. An error is signaled if `process` does not identify a process.

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 it 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.

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 37.4 [Asynchronous Processes], page 710.

**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` [Function]

This function kills the process `process` by sending the signal `SIGKILL`. This signal kills the subprocess immediately, and cannot be handled by the subprocess.

**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-b` or `C-\`) when you are not inside Emacs.

**stop-process** &optional `process` `current-group` [Function]

This function stops the process `process` by sending the signal `SIGTSTP`. Use `continue-process` to resume its execution.

Outside of Emacs, on systems with job control, the "stop character" (usually `C-z`) normally sends this signal. 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* by sending it the signal SIGCONT. This presumes that *process* was stopped previously.

**signal-process** *process signal* [Function]

This function sends a signal to process *process*. The argument *signal* specifies which signal to send; it should be an integer.

The *process* argument can be a system process ID; that allows you to send signals to processes that are not children of Emacs.

## 37.9 Receiving Output from Processes

There are two ways to receive the output that a subprocess writes to its standard output stream. The output can be inserted in a buffer, which is called the associated buffer of the process, or a function called the *filter function* can be called to act on the output. If the process has no buffer and no filter function, its output is discarded.

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, in `sit-for` and `sleep-for` (see Section 21.9 [Waiting], page 337), and in `accept-process-output` (see Section 37.9.4 [Accepting Output], page 721). 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.

It is impossible to separate the standard output and standard error streams of the subprocess, because Emacs normally spawns the subprocess inside a pseudo-TTY, and a pseudo-TTY has only one output channel. If you want to keep the output to those streams separate, you should redirect one of them to a file—for example, by using an appropriate shell command.

### 37.9.1 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.

Unless the process has a filter function (see Section 37.9.2 [Filter Functions], page 718), its output is inserted in the associated buffer. 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.

**process-buffer** *process* [Function]

This function returns the associated buffer of the process *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.

Insertion of process output in a buffer uses this marker to decide where to insert, and updates it to point after the inserted text. That is why successive batches of output are inserted consecutively.

Filter functions normally should use this marker in the same fashion as is done by direct insertion of output in the buffer. A good example of a filter function that uses `process-mark` is found at the end of the following section.

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.

**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 37.8 [Signals to Processes], page 715).

## 37.9.2 Process Filter Functions

A process *filter function* is a function that receives the standard output from the associated process. If a process has a filter, then *all* output from that process is passed to the filter. The process buffer is used directly for output from the process only when there is no filter.

The filter function can only be called when Emacs is waiting for something, because process output arrives only at such times. Emacs waits when reading terminal input, in `sit-for` and `sleep-for` (see Section 21.9 [Waiting], page 337), and in `accept-process-output` (see Section 37.9.4 [Accepting Output], page 721).

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 21.10 [Quitting], page 338.

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**, the error-catching is turned off. This makes it possible to use the Lisp debugger to debug the filter function. See Section 18.1 [Debugger], page 237.

Many filter functions sometimes or always insert the text in the process's buffer, mimicking the actions of Emacs when there is no filter. Such filter functions need to use **set-buffer** in order to be sure to insert in that buffer. To avoid setting the current buffer semipermanently, these filter functions must save and restore the current buffer. They should also 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)
  (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))))))
```

The reason to use **with-current-buffer**, rather than using **save-excursion** to save and restore the current buffer, is so as to preserve the change in point made by the second call to **goto-char**.

To make the filter force the process buffer to be visible whenever new text arrives, insert the following line 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** and call **goto-char** unconditionally.

In earlier Emacs versions, every filter function that did regular expression searching or matching had to explicitly save and restore the match data. Now Emacs does this automatically for filter functions; they never need to do it explicitly. See Section 34.6 [Match Data], page 676.

A filter function 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. The expression **(buffer-name (process-buffer process))** returns **nil** if the buffer is dead.

The output to the function 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.

**set-process-filter** *process filter* [Function]

This function gives *process* the filter function *filter*. If *filter* is `nil`, it gives the process no filter.

**process-filter** *process* [Function]

This function returns the filter function of *process*, or `nil` if it has none.

Here is an example of 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[8] % "
"FINAL-W87-SHORT.MSS      backup.ctl          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"
")
```

### 37.9.3 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 37.6 [Process Information], page 712). Otherwise, the coding system comes from `coding-system-for-read`, if that is non-`nil`; or else from the defaulting mechanism (see Section 33.10.5 [Default Coding Systems], page 653).

**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 multibyte flag. If the flag is non-`nil`, Emacs decodes the output according to the process output coding system to

produce a multibyte string, and passes that to the process. If the flag is `nil`, Emacs puts the output into a unibyte string, with no decoding, and passes that.

When you create a process, the filter multibyte flag takes its initial value from `default-enable-multibyte-characters`. If you want to change the flag later on, use `set-process-filter-multibyte`.

`set-process-filter-multibyte process multibyte` [Function]  
 This function sets the filter multibyte flag of `process` to `multibyte`.

`process-filter-multibyte-p process` [Function]  
 This function returns the filter multibyte flag of `process`.

### 37.9.4 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` [Function]  
`just-this-one`

This function allows Emacs to read pending output from processes. The output is inserted in the associated buffers or given to their filter functions. If `process` is non-`nil` then this function does not return until some output has been received from `process`.

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, whether or not there has been any subprocess output.

The argument `millisec` is semi-obsolete nowadays because `seconds` can be a floating point number 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 did get some output, or `nil` if the timeout expired before output arrived.

### 37.10 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.

The string describing the event looks like one of the following:

- "finished\n".
- "exited abnormally with code `exitcode`\n".
- "`name-of-signal`\n".
- "`name-of-signal` (core dumped)\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 them 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 21.9 [Waiting], page 337), or `accept-process-output` (see Section 37.9.4 [Accepting Output], page 721). 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 21.10 [Quitting], page 338.

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`, the error-catching is turned off. This makes it possible to use the Lisp debugger to debug the sentinel. See Section 18.1 [Debugger], page 237.

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.

In earlier Emacs versions, every sentinel that did regular expression searching or matching had to explicitly save and restore the match data. Now Emacs does this automatically for sentinels; they never need to do it explicitly. See Section 34.6 [Match Data], page 676.

**set-process-sentinel** *process sentinel*

[Function]

This function associates *sentinel* with *process*. If *sentinel* is `nil`, then the process will have no sentinel. The default behavior when there is no sentinel is to insert a message in the process's buffer when the process status changes.

Changes in process sentinel 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*, or `nil` if it has none.

`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, `nil` if it was not.

## 37.11 Querying Before Exit

When Emacs exits, it terminates all its subprocesses by sending them the `SIGHUP` signal. 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*.

```
; ; Don't query about the shell process
(set-process-query-on-exit-flag (get-process "shell") nil)
⇒ t
```

`process-kill-without-query` *process* &**optional** *do-query* [Function]

This function clears the query flag of *process*, so that Emacs will not query the user on account of that process.

Actually, the function does more than that: it returns the old value of the process's query flag, and sets the query flag to *do-query*. Please don't use this function to do those things any more—please use the newer, cleaner functions `process-query-on-exit-flag` and `set-process-query-on-exit-flag` in all but the simplest cases. The only way you should use `process-kill-without-query` nowadays is like this:

```
; ; Don't query about the shell process
(process-kill-without-query (get-process "shell"))
```

## 37.12 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.

The return value of **tq-enqueue** itself is not meaningful.

**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 37.9.2 [Filter Functions], page 718.

### 37.13 Network Connections

Emacs Lisp programs can open stream (TCP) and datagram (UDP) network connections 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, so it 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 37.16 [Low-Level Network], page 727, for details. You can also use the **open-network-stream** function described below.

You can distinguish process objects representing network connections and servers from those representing subprocesses with the `process-status` function. The possible status values for network connections are `open`, `closed`, `connect`, and `failed`. For a network server, the status is always `listen`. None of those values is possible for a real subprocess. See Section 37.6 [Process Information], page 712.

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.) 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.

`open-network-stream name buffer-or-name host service` [Function]

This function opens a TCP connection, 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-or-name` argument is the buffer to associate with the connection. Output from the connection is inserted in the buffer, unless you specify a filter function to handle the output. If `buffer-or-name` 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).

`process-contact process &optional key` [Function]

This function returns information about how a network process was set up. For a connection, when `key` is `nil`, it returns (`hostname service`) which specifies what you connected to.

If `key` is `t`, the value is the complete status information for the connection or server; that is, the list of keywords and values specified in `make-network-process`, except that some of the values represent the current status instead of what you specified:

- `:buffer` The associated value is the process buffer.
- `:filter` The associated value is the process filter function.
- `:sentinel` The associated value is the process sentinel function.
- `: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`.

If `key` is a keyword, the function returns the value corresponding to that keyword.

For an ordinary child process, this function always returns `t`.

## 37.14 Network Servers

You create a server by calling `make-network-process` 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' name with a client identification string. The client identification string for an IPv4 connection looks like '`<a.b.c.d:p>`'. 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's filter is `non-nil`, 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 by Emacs, but it is passed to the `log` function, which can 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 info 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' plist is initialized from the server's plist.

## 37.15 Datagrams

A datagram connection communicates with individual packets rather than streams of data. Each call to `process-send` sends one datagram packet (see Section 37.7 [Input to Processes], page 714), 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*.

## 37.16 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`.

### 37.16.1 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:

**: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. Both connections and servers can be of either type.

**: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.

**: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 that translates to a port number, or an integer specifying 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.

**:local *local-address***

For a server process, *local-address* is the address to listen on. It overrides *family*, *host* and *service*, and you may 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*, and you may 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.

**:stop *stopped***

Start the network connection or server in the ‘stopped’ state if *stopped* is non-*nil*.

**: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 37.11 [Query Before Exit], page 723.

`:filter filter`

Initialize the process filter to *filter*.

`:filter-multibyte bool`

If *bool* is non-**nil**, strings given to the process filter are multibyte, otherwise they are unibyte. If you don't specify this keyword at all, the default is that the strings are multibyte if **default-enable-multibyte-characters** is non-**nil**.

`: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.

### 37.16.2 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. 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. 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` [Function]

This function sets or modifies a network option for network process `process`. See `make-network-process` for details of options `option` and their corresponding values `value`.

The current setting of an option is available via the `process-contact` function.

### 37.16.3 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 the first form is `t` if it works to specify `keyword` with value `value` in `make-network-process`. The result of the second form is `t` if `keyword` is supported by `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)
```

Here are some of the options you can test in this way.

```
:bindtodevice
:broadcast
:dontroute
:keepalive
:linger
:oobinline
:priority
:reuseaddr
```

That particular network option is supported by `make-network-process` and `set-network-process-option`.

## 37.17 Misc Network Facilities

These additional functions are useful for creating and operating on network connections.

### `network-interface-list`

[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 (`name . address`). `address` has the same form as the `local-address` and `remote-address` arguments to `make-network-process`.

### `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`).

- |                      |   |
|----------------------|---|
| <code>addr</code>    | The Internet protocol address.                            |
| <code>bcast</code>   | The broadcast address.                                    |
| <code>netmask</code> | The network mask.   |
| <code>hwaddr</code>  | The layer 2 address (Ethernet MAC address, for instance). |
| <code>flags</code>   | The current flags of the interface.                       |

### `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.

## 37.18 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.

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*.

### 37.18.1 Describing Data Layout

To control unpacking and packing, you write a *data layout specification*, a special nested list describing named and typed *fields*. This specification controls length of each field to be processed, and how to pack or unpack it. We normally keep bindat specs in variables whose names end in ‘`-bindat-spec`’; that kind of name is automatically recognized as “risky.”

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:

<code>u8</code>	
<code>byte</code>	Unsigned byte, with length 1.
<code>u16</code>	
<code>word</code>	
<code>short</code>	Unsigned integer in network byte order, with length 2.
<code>u24</code>	Unsigned integer in network byte order, with length 3.
<code>u32</code>	
<code>dword</code>	
<code>long</code>	Unsigned integer in network byte order, with length 4. Note: These values may be limited by Emacs’ integer implementation limits.
<code>u16r</code>	
<code>u24r</code>	
<code>u32r</code>	Unsigned integer in little endian order, with length 2, 3 and 4, respectively.
<code>str len</code>	String of length <code>len</code> .
<code>strz len</code>	Zero-terminated string, in a fixed-size field with length <code>len</code> .
<code>vec len [type]</code>	Vector of <code>len</code> elements of type <code>type</code> , or bytes if not <code>type</code> is specified. The <code>type</code> is any of the simple types above, or another vector specified as a list ( <code>vec len [type]</code> ).
<code>ip</code>	Four-byte vector representing an Internet address. For example: <code>[127 0 0 1]</code> for localhost.
<code>bits len</code>	List of set bits in <code>len</code> bytes. The bytes are taken in big endian order and the bits are numbered starting with $8 * \text{len} - 1$ and ending with zero. For example: <code>bits 2</code> unpacks <code>#x28 #x1c</code> to <code>(2 3 4 11 13)</code> and <code>#x1c #x28</code> to <code>(3 5 10 11 12)</code> .

**(eval form)**

*form* is a Lisp expression evaluated at the moment the field is unpacked or packed. The result of the evaluation should be one of the above-listed type specifications.

For a fixed-size field, the length *len* is given as an integer specifying the number of bytes in the field.

When the length of a field is not fixed, it typically depends on the value of a preceding field. In this case, the length *len* can be given either as a list (*name* ...) identifying a field *name* in the format specified for `bindat-get-field` below, or by an expression (`eval form`) where *form* should evaluate to an integer, specifying the field length.

A field specification generally has the form ([*name*] *handler*). The square braces indicate that *name* is optional. (Don't use names that are symbols meaningful as type specifications (above) or handler specifications (below), since that would be ambiguous.) *name* can be a symbol or the expression (`eval form`), in which case *form* should evaluate to a symbol.

*handler* describes how to unpack or pack the field and can be one of the following:

**type** Unpack/pack this field according to the type specification *type*.

**eval form**

Evaluate *form*, a Lisp expression, for side-effect only. If the field name is specified, the value is bound to that field name.

**fill len** Skip *len* bytes. In packing, this leaves them unchanged, which normally means they remain zero. In unpacking, this means they are ignored.

**align len**

Skip to the next multiple of *len* bytes.

**struct spec-name**

Process *spec-name* as a sub-specification. This describes a structure nested within another structure.

**union form (tag spec)...**

Evaluate *form*, a Lisp expression, find the first *tag* that matches it, and process its associated data layout specification *spec*. Matching can occur in one of three ways:

- If a *tag* has the form (`eval expr`), evaluate *expr* with the variable *tag* dynamically bound to the value of *form*. A non-nil result indicates a match.
- *tag* matches if it is `equal` to the value of *form*.
- *tag* matches unconditionally if it is `t`.

**repeat count field-specs...**

Process the *field-specs* recursively, in order, then repeat starting from the first one, processing all the specs *count* times overall. The *count* is given using the same formats as a field length—if an `eval` form is used, it is evaluated just once. For correct operation, each spec in *field-specs* must include a name.

For the (`eval form`) forms used in a bindat specification, the *form* can access and update these dynamically bound variables during evaluation:

<code>last</code>	Value of the last field processed.
<code>bindat-raw</code>	The data as a byte array.
<code>bindat-idx</code>	Current index (within <code>bindat-raw</code> ) for unpacking or packing.
<code>struct</code>	The alist containing the structured data that have been unpacked so far, or the entire structure being packed. You can use <code>bindat-get-field</code> to access specific fields of this structure.
<code>count</code>	
<code>index</code>	Inside a <code>repeat</code> block, these contain the maximum number of repetitions (as specified by the <code>count</code> parameter), and the current repetition number (counting from 0). Setting <code>count</code> to zero will terminate the inner-most repeat block after the current repetition has completed.

### 37.18.2 Functions to Unpack and Pack Bytes

In the following documentation, *spec* refers to a data layout specification, `bindat-raw` to a byte array, and *struct* to an alist representing unpacked field data.

**`bindat-unpack spec bindat-raw &optional bindat-idx`** [Function]

This function unpacks data from the unibyte string or byte array `bindat-raw` according to *spec*. Normally this starts unpacking at the beginning of the byte array, but if `bindat-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, if *name* is `(a b 2 c)`, that means to find field *c* in the third element of subfield *b* of field *a*. (This corresponds to `struct.a.b[2].c` in C.)

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 spec struct`** [Function]

This function returns the total length of the data in *struct*, according to *spec*.

**bindat-pack** *spec struct &optional bindat-raw bindat-idx* [Function]

This function returns a byte array packed according to *spec* from the data in the alist *struct*. Normally it creates and fills a new byte array starting at the beginning. However, if *bindat-raw* is non-nil, it specifies a pre-allocated unibyte string or vector to pack into. If *bindat-idx* is non-nil, it specifies the starting offset for packing into *bindat-raw*.

When pre-allocating, you should make sure (*length bindat-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"
```

### 37.18.3 Examples of Byte Unpacking and Packing

Here is a complete example of byte unpacking and packing:

```
(defvar fcookie-index-spec
'((:version u32)
 (:count u32)
 (:longest u32)
 (:shortest u32)
 (:flags u32)
 (:delim u8)
 (:ignored fill 3)
 (:offset repeat (:count)
 (:foo u32)))
 "Description of a fortune cookie index file's contents.")

(defun fcookie (cookies &optional index)
  "Display a random fortune cookie from file COOKIES.
Optional second arg INDEX specifies the associated index
filename, which is by default constructed by appending
\".dat\" to COOKIES.  Display cookie text in possibly
new buffer \"*Fortune Cookie: BASENAME*\" where BASENAME
is COOKIES without the directory part."
  (interactive "fCookies file: ")
  (let* ((info (with-temp-buffer
                  (insert-file-contents-literally
                   (or index (concat cookies ".dat"))))
         (bindat-unpack fcookie-index-spec
                      (buffer-string))))
    (sel (random (bindat-get-field info :count)))
    (beg (cdar (bindat-get-field info :offset sel)))
    (end (or (cdar (bindat-get-field info
                                         :offset (1+ sel)))
              (nth 7 (file-attributes cookies)))))
```

```
(switch-to-buffer
  (get-buffer-create
    (format "*Fortune Cookie: %s*"
           (file-name-nondirectory cookies))))
  (erase-buffer)
  (insert-file-contents-literally
    cookies nil beg (- end 3)))

(defun fcookie-create-index (cookies &optional index delim)
  "Scan file COOKIES, and write out its index file.
Optional second arg INDEX specifies the index filename,
which is by default constructed by appending \".\dat\" to
COOKIES. Optional third arg DELIM specifies the unibyte
character which, when found on a line of its own in
COOKIES, indicates the border between entries."
  (interactive "fCookies file: ")
  (setq delim (or delim ?%))
  (let ((delim-line (format "\n%c\n" delim)))
    (count 0)
    (max 0)
    (min p q len offsets)
    (unless (= 3 (string-bytes delim-line))
      (error "Delimiter cannot be represented in one byte")))
    (with-temp-buffer
      (insert-file-contents-literally cookies)
      (while (and (setq p (point))
                  (search-forward delim-line (point-max) t)
                  (setq len (- (point) 3 p)))
        (setq count (1+ count)
              max (max max len)
              min (min (or min max) len)
              offsets (cons (1- p) offsets))))
    (with-temp-buffer
      (set-buffer-multibyte nil)
      (insert
        (bindat-pack
          fcookie-index-spec
          '(((:version . 2)
            (:count . ,count)
            (:longest . ,max)
            (:shortest . ,min)
            (:flags . 0)
            (:delim . ,delim)
            (:offset . ,(mapcar (lambda (o)
                                  (list (cons :foo o)))
                                 (nreverse offsets)))))))
    (let ((coding-system-for-write 'raw-text-unix))
```

```
(write-file (or index (concat cookies ".dat"))))))
```

Following is an example of defining and unpacking a complex structure. Consider the following C structures:

```
struct header {
    unsigned long    dest_ip;
    unsigned long    src_ip;
    unsigned short   dest_port;
    unsigned short   src_port;
};

struct data {
    unsigned char    type;
    unsigned char    opcode;
    unsigned short   length; /* In network byte order */
    unsigned char    id[8]; /* null-terminated string */
    unsigned char    data[/* (length + 3) & ~3 */];
};

struct packet {
    struct header    header;
    unsigned long    counters[2]; /* In little endian order */
    unsigned char    items;
    unsigned char    filler[3];
    struct data     item[/* items */];
};

};
```

The corresponding data layout specification:

```
(setq header-spec
      '((dest-ip ip)
        (src-ip ip)
        (dest-port u16)
        (src-port u16)))

(setq data-spec
      '((type u8)
        (opcode u8)
        (length u16) ;; network byte order
        (id strz 8)
        (data vec (length))
        (align 4)))

(setq packet-spec
      '((header struct header-spec)
        (counters vec 2 u32r) ;; little endian order
        (items u8)
        (fill 3))
```

```
(item      repeat (items)
      (struct data-spec))))
```

A binary data representation:

```
(setq binary-data
  [ 192 168 1 100 192 168 1 101 01 28 21 32
    160 134 1 0 5 1 0 0 2 0 0 0
    2 3 0 5 ?A ?B ?C ?D ?E ?F 0 0 1 2 3 4 5 0 0 0
    1 4 0 7 ?B ?C ?D ?E ?F ?G 0 0 6 7 8 9 10 11 12 0 ])
```

The corresponding decoded structure:

```
(setq decoded (bindat-unpack packet-spec binary-data))
⇒
(header
  (dest-ip . [192 168 1 100])
  (src-ip . [192 168 1 101])
  (dest-port . 284)
  (src-port . 5408))
(counters . [100000 261])
(items . 2)
(item ((data . [1 2 3 4 5])
        (id . "ABCDEF")
        (length . 5)
        (opcode . 3)
        (type . 2))
      ((data . [6 7 8 9 10 11 12])
        (id . "BCDEFG")
        (length . 7)
        (opcode . 4)
        (type . 1))))
```

Fetching data from this structure:

```
(bindat-get-field decoded 'item 1 'id)
⇒ "BCDEFG"
```

## 38 Emacs Display

This chapter describes a number of features related to the display that Emacs presents to the user.

### 38.1 Refreshing the Screen

The function `redraw-frame` clears and redisplays the entire contents of a given frame (see Chapter 29 [Frames], page 529). This is useful if the screen is corrupted.

`redraw-frame frame` [Function]  
This function clears and redisplays frame *frame*.

Even more powerful is `redraw-display`:

`redraw-display` [Command]  
This function clears and redisplays all visible frames.

This function calls for redisplay of certain windows, the next time redisplay is done, but does not clear them first.

`force-window-update &optional object` [Function]  
This function forces some or all windows to be updated on next redisplay. If *object* is a window, it forces redisplay of that window. If *object* is a buffer or buffer name, it forces redisplay of all windows displaying that buffer. If *object* is `nil` (or omitted), it forces redisplay of all windows.

Processing user input takes absolute priority over redisplay. If you call these functions when input is available, they do nothing immediately, but a full redisplay does happen eventually—after all the input has been processed.

Normally, suspending and resuming Emacs 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` [Variable]  
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`.

### 38.2 Forcing Redisplay

Emacs redisplay normally stops if input arrives, and does not happen at all if input is available before it starts. Most of the time, this is exactly what you want. However, you can prevent preemption by binding `redisplay-dont-pause` to a non-`nil` value.

`redisplay-preemption-period` [Variable]  
This variable specifies how many seconds Emacs waits between checks for new input during redisplay. (The default is 0.1 seconds.) If input has arrived when Emacs

checks, it pre-empts redisplay and processes the available input before trying again to redisplay.

If this variable is `nil`, Emacs does not check for input during redisplay, and redisplay cannot be preempted by input.

This variable is only obeyed on graphical terminals. For text terminals, see Section 39.13 [Terminal Output], page 834.

#### **redisplay-dont-pause**

[Variable]

If this variable is non-`nil`, pending input does not prevent or halt redisplay; redisplay occurs, and finishes, regardless of whether input is available.

#### **redisplay &optional force**

[Function]

This function performs an immediate redisplay provided there are no pending input events. This is equivalent to `(sit-for 0)`.

If the optional argument `force` is non-`nil`, it forces an immediate and complete redisplay even if input is available.

Returns `t` if redisplay was performed, or `nil` otherwise.

### 38.3 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 32.11 [Filling], page 599.

On a graphical display, tiny arrow images in the window fringes indicate truncated and continued lines (see Section 38.13 [Fringes], page 776). On a text terminal, 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 38.21 [Display Tables], page 807).

#### **truncate-lines**

[User Option]

This buffer-local variable controls how Emacs displays lines that extend beyond the right edge of the window. The default is `nil`, which specifies continuation. If the value is non-`nil`, then these lines are truncated.

If the variable `truncate-partial-width-windows` is non-`nil`, then truncation is always used for side-by-side windows (within one frame) regardless of the value of `truncate-lines`.

#### **default-truncate-lines**

[User Option]

This variable is the default value for `truncate-lines`, for buffers that do not have buffer-local values for it.

#### **truncate-partial-width-windows**

[User Option]

This variable controls display of lines that extend beyond the right edge of the window, in side-by-side windows (see Section 28.2 [Splitting Windows], page 498). If it is non-`nil`, these lines are truncated; otherwise, `truncate-lines` says what to do with them.

When horizontal scrolling (see Section 28.13 [Horizontal Scrolling], page 518) is in use in a window, that forces truncation.

If your buffer contains *very* long lines, and you use continuation to display them, just thinking about them can make Emacs redisplay slow. The column computation and indentation functions also become slow. Then you might find it advisable to set `cache-long-line-scans` to `t`.

#### `cache-long-line-scans`

[Variable]

If this variable is `non-nil`, various indentation and motion functions, and Emacs redisplay, cache the results of scanning the buffer, and consult the cache to avoid rescanning regions of the buffer unless they are modified.

Turning on the cache slows down processing of short lines somewhat.

This variable is automatically buffer-local in every buffer.

## 38.4 The Echo Area

The echo area is used for displaying error messages (see Section 10.5.3 [Errors], page 127), 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. The *GNU Emacs Manual* specifies the rules for resolving conflicts between the echo area and the minibuffer for use of that screen space (see section “The Minibuffer” in *The GNU Emacs Manual*).

You can write output in the echo area by using the Lisp printing functions with `t` as the stream (see Section 19.5 [Output Functions], page 273), or explicitly.

### 38.4.1 Displaying Messages in the Echo Area

This section describes the functions for explicitly producing echo area messages. Many other Emacs features display messages there, too.

#### `message format-string &rest arguments`

[Function]

This function displays a message in the echo area. The argument `format-string` is similar to a C language `printf` format string. See `format` in Section 4.7 [Formatting Strings], page 56, for the details on the conversion specifications. `message` returns the constructed string.

In batch mode, `message` prints the message text on the standard error stream, followed by a newline.

If `format-string`, or strings among the `arguments`, have `face` text properties, these affect the way the message is displayed.

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 "Minibuffer depth is %d."
        (minibuffer-depth))
  → Minibuffer depth is 0.
  ⇒ "Minibuffer depth is 0."
```

```
----- Echo Area -----
Minibuffer depth is 0.
----- 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).

**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 21.4 [Command Loop Info], page 312) 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 20.7 [Yes-or-No Queries], page 296.)

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 not-this-window 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 *not-this-window* 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.

### 38.4.2 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.

Functions listed in this section provide simple and efficient way of reporting operation progress. 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 min-value max-value &optional current-value min-change min-time* [Function]

This function creates and returns a *progress reporter*—an object you will use as an argument for all other functions listed here. 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** before calling this function. *min-value* and *max-value* arguments stand for starting and final states of your operation. For instance, if you scan a buffer, they should be the results of **point-min** and **point-max** correspondingly. It is required that *max-value* is greater than *min-value*. If you create *progress reporter* when some part of the operation has already been completed, then specify *current-value* argument. But normally you should omit it or set it to **nil**—it will default to *min-value* then.

Remaining arguments control the rate of echo area updates. *Progress reporter* will wait for at least *min-change* more percents of the operation to be completed before printing next message. *min-time* specifies the minimum time in seconds to pass between successive prints. It can be fractional. Depending on Emacs and system capabilities, *progress reporter* may or may not respect this last argument or do it with varying precision. Default value for *min-change* is 1 (one percent), for *min-time*—0.2 (seconds.)

This function calls **progress-reporter-update**, so the first message is printed immediately.

**progress-reporter-update** *reporter value* [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**.

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 value &optional new-message* [Function]

This function is similar to `progress-reporter-update` except that it prints a message in the echo area unconditionally.

The first two arguments 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]*) *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.

You can rewrite the example in the beginning of this node using this macro this way:

```
(dotimes-with-progress-reporter
  (k 500)
  "Collecting some mana for Emacs..."
  (sit-for 0.01))
```

### 38.4.3 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`.

**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 ...))
```

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” means 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” means 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 ‘...’.

#### 38.4.4 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`

[Variable]

This variable determines how much time should elapse before command characters echo. Its value must be an integer or floating point number, which 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. But if the variable `message-truncate-lines` is `non-nil`, the echo area does not resize, and the message is truncated to fit it, as in Emacs 20 and before.

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 20.14 [Minibuffer Misc], page 302.

### 38.5 Reporting Warnings

Warnings are a facility for a program to inform the user of a possible problem, but continue running.

### 38.5.1 Warning Basics

Every warning has a textual message, which explains the problem for the user, and a *severity level* which is a symbol. Here are the possible severity levels, in order of decreasing severity, and their meanings:

- :emergency** A problem that will seriously impair Emacs operation soon if you do not attend to it promptly.
- :error** A report of data or circumstances that are inherently wrong.
- :warning** A report of 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 you are debugging.

When your program encounters invalid input data, it can either signal a Lisp error by calling `error` or `signal` or report a warning with severity `:error`. Signaling a Lisp error is the easiest thing to do, but it means the program cannot continue processing. If you want to take the trouble to implement a way to continue processing despite the bad data, then reporting a warning of severity `:error` is the right way to inform 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; it could show the message to the user by reporting it as a warning.)

Each warning has a *warning type* to classify it. The type is a list of symbols. The first symbol should be the custom group that you use for the program's user options. For example, byte compiler warnings use the warning type (`bytecomp`). You can also subcategorize the warnings, if you wish, by using more symbols in the list.

**display-warning** *type message &optional level buffer-name* [Function]

This function reports a warning, using *message* as the message and *type* as the warning type. *level* should be the severity level, with `:warning` being the default.

*buffer-name*, if non-`nil`, specifies the name of the buffer for logging the warning. By default, it is '`*Warnings*`'.

**lwarn** *type level message &rest args* [Function]

This function reports a warning using the value of `(format message args...)` as the message. In other respects it is equivalent to `display-warning`.

**warn** *message &rest args* [Function]

This function reports a warning using the value of `(format message args...)` as the message, `(emacs)` as the 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.

### 38.5.2 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.

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 current, and the function can insert text in 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 as the entry (this value need not be an actual member of `warning-levels`). 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* then there is no function to call.

**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 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 current. The function can insert text which will serve as a header for the series of warnings.

Once a series has begun, the value 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 each warning's text.

**warning-type-format**

[Variable]

This variable specifies the format for displaying the warning type in the warning message. 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 "" then the warning type won't appear at all.

### 38.5.3 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. The default is `:warning`, which means to immediately display all warnings except `:debug` warnings.

#### `warning-minimum-log-level`

[User Option]

This user option specifies the minimum severity level that should be logged in the warnings buffer. The default is `:warning`, which means to log all warnings except `:debug` warnings.

#### `warning-suppress-types`

[User Option]

This list specifies which warning types should not be displayed immediately for the user. Each element of the list should be a list of symbols. If its elements match the first elements in a warning type, then that warning is not displayed immediately.

#### `warning-suppress-log-types`

[User Option]

This list specifies which warning types should not be logged in the warnings buffer. Each element of the list should be a list of symbols. If it matches the first few elements in a warning type, then that warning is not logged.

## 38.6 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 32.19 [Text Properties], page 615) or a property of an overlay (see Section 38.9 [Overlays], page 754). Cursor motion also partly ignores these characters; if the command loop finds point within them, it moves point to the other side of them.

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.

<b>t</b>	A character is invisible if its <code>invisible</code> property is non- <code>nil</code> . This is the default.
a list	Each element of the list specifies a criterion for invisibility; if a character's <code>invisible</code> property fits any one of these criteria, the character is invisible. The list can have two kinds of elements:
<code>atom</code>	A character is invisible if its <code>invisible</code> property value is <code>atom</code> or if it is a list with <code>atom</code> as a member.
<code>(atom . t)</code>	A character is invisible if its <code>invisible</code> property value is <code>atom</code> or if it is a list with <code>atom</code> as a member. 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 overlays:  

(remove-from-invisibility-spec '(my-symbol . t))  

; ; Or respectively:  

(remove-from-invisibility-spec 'my-symbol)
```

Ordinarily, functions that operate on text or move point do not care whether the text is invisible. The user-level line motion commands explicitly ignore invisible newlines if `line-move-ignore-invisible` is non-`nil` (the default), but only because they are explicitly programmed to do so.

However, if a command ends with point inside or immediately before invisible text, the main editing loop moves point further forward or further backward (in the same direction that the command already moved it) until that condition is no longer true. Thus, if the

command moved point back into an invisible range, Emacs moves point back to the beginning of that range, and then back one more character. If the command moved point forward into an invisible range, Emacs moves point forward up to the first visible character that follows the invisible text.

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.

## 38.7 Selective Display

*Selective display* refers to a pair of related features for hiding certain lines on the screen.

The first variant, explicit selective display, is designed for use in a Lisp program: it controls which lines are hidden by altering the text. This kind of hiding in some ways resembles the effect of the `invisible` property (see Section 38.6 [Invisible Text], page 748), but the two features are different and do not work the same way.

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 an 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`

[Variable]

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 38.21 [Display Tables], page 807.

## 38.8 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 forms...* [Special Form]

This function executes *forms* 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. Finally, the buffer is displayed in some window, but not selected.

If the *forms* 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 864, 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 *forms*. Output using the Lisp output functions within *forms* 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 19.5 [Output Functions], page 273.

Several hooks are available for customizing the behavior of this construct; they are listed below.

The value of the last form in *forms* 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** [Variable]

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. This hook is normally set up with a function to make the buffer read only, and find function names and variable names in it, provided the major mode is Help mode.

**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 38.9.2 [Overlay Properties], page 756.

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 -----
```

## 38.9 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 overlay.

An overlay uses markers to record its beginning and end; thus, 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.

### 38.9.1 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 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.

The arguments *front-advance* and *rear-advance* specify the marker insertion type for the start of the overlay and for the end of the overlay, respectively. See Section 31.5 [Marker Insertion Types], page 576. 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 *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*. 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 deleted, it goes into the current buffer.

The return value is *overlay*.

This is the only valid way to change the endpoints of an overlay. Do not try modifying the markers in the overlay by hand, as that fails to update other vital data structures and can cause some overlays to be “lost.”

**remove-overlays** *&optional start end name value* [Function]

This function removes all the overlays between *start* and *end* whose property *name* has the value *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.

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
```

Emacs stores the overlays of each buffer in two lists, divided around an arbitrary “center position.” One list extends backwards through the buffer from that center position, and the other extends forwards from that center position. The center position can be anywhere in the buffer.

#### **overlay-recenter pos** [Function]

This function recenters the overlays of the current buffer around position *pos*. That makes overlay lookup faster for positions near *pos*, but slower for positions far away from *pos*.

A loop that scans the buffer forwards, creating overlays, can run faster if you do (*overlay-recenter (point-max)*) first.

### 38.9.2 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 32.19 [Text Properties], page 615, 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.

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 32.19.1 [Examining Properties], page 615.

Many overlay properties have special meanings; here is a table of them:

**priority** This property's value (which should be a nonnegative integer number) determines the priority of the overlay. The priority matters when two or more overlays cover the same character and both specify the same property; the one whose **priority** value is larger takes priority over the other. For the **face** property, the higher priority value does not completely replace the other; instead, its face attributes override the face attributes of the lower priority **face** property. Currently, all overlays take priority over text properties. Please avoid using negative priority values, as we have not yet decided just what they should mean.

**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 way text is displayed—for example, which font and which colors. See Section 38.12 [Faces], page 762, for more information.

In the simplest case, the value is a face name. It can also be a list; then each element can be any of these possibilities:

- A face name (a symbol or string).
- A property list of face attributes. This has the form (*keyword value ...*), where each *keyword* is a face attribute name and *value* is a meaningful value for that attribute. With this feature, you do not need to create a face each time you want to specify a particular attribute for certain text. See Section 38.12.2 [Face Attributes], page 765.

- A cons cell, either of the form (`foreground-color . color-name`) or (`background-color . color-name`). These elements specify just the foreground color or just the background color.  
`(foreground-color . color-name)` has the same effect as (`:foreground color-name`); likewise for the background.

**mouse-face**

This property is used instead of `face` when the mouse is within the range of the overlay.

**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 38.15 [Display Property], page 783.

**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 in the tooltip window. For details see [Text help-echo], page 621.

**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.)

If these functions modify the buffer, they should bind `inhibit-modification-hooks` to `t` around doing so, to avoid confusing the internal mechanism that calls these hooks.

Text properties also support the `modification-hooks` property, but the details are somewhat different (see Section 32.19.4 [Special Properties], page 620).

**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 38.6 [Invisible Text], page 748, for details.

**intangible**

The **intangible** property on an overlay works just like the **intangible** text property. See Section 32.19.4 [Special Properties], page 620, for details.

**isearch-open-invisible**

This property tells incremental search how to make an invisible overlay visible, permanently, if the final match overlaps it. See Section 38.6 [Invisible Text], page 748.

**isearch-open-invisible-temporary**

This property tells incremental search how to make an invisible overlay visible, temporarily, during the search. See Section 38.6 [Invisible Text], page 748.

**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.

**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.

**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 a **non-nil** **evaporate** property, that deletes it immediately.

**local-map**

If this property is **non-nil**, it specifies a keymap for a portion of the text. The property's value replaces the buffer's local map, when the character after point is within the overlay. See Section 22.7 [Active Keymaps], page 353.

**keymap**

The **keymap** property is similar to **local-map** but overrides the buffer's local map (and the map specified by the **local-map** property) rather than replacing it.

### 38.9.3 Searching for Overlays

**overlays-at pos**

[Function]

This function returns a list of all the overlays that cover the character at position *pos* in the current buffer. The list 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*. “Overlap” means that at least one character is contained within the overlay and also contained within the specified region; however, empty overlays are included in the result if they are located at *beg*, or strictly between *beg* and *end*.

**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 32.19.3 [Property Search], page 618). 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))))
```

## 38.10 Width

Since not all characters have the same width, these functions let you check the width of a character. See Section 32.17.1 [Primitive Indent], page 610, and Section 30.2.5 [Screen Lines], page 564, 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 and the selected window.

**string-width** *string*

[Function]

This function returns the width in columns of the string *string*, if it were displayed in the current buffer and the selected window.

**truncate-string-to-width** *string width &optional start-column padding ellipsis*

[Function]

This function returns the part of *string* that fits within *width* columns, as a new string.

If *string* does not reach *width*, then the result ends where *string* ends. If one multi-column character in *string* extends across the column *width*, that character is not included in the result. Thus, the result can fall short of *width* but cannot go beyond it.

The optional argument *start-column* specifies the starting column. If this is non-*nil*, then the first *start-column* columns of the string are omitted from the value. If one multi-column character in *string* extends across the column *start-column*, that character is not included.

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 used at the end of the result if it falls short of *width*. It is also used at the beginning of the result if one 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 *str* (including any padding) if it extends beyond *end-column*, unless the display width of *str* is equal to or less than the display width of *ellipsis*. If *ellipsis* is non-*nil* and not a string, it stands for "...".

```
(truncate-string-to-width "\tab\t" 12 4)
⇒ "ab"
(truncate-string-to-width "\tab\t" 12 4 ?\s)
⇒ "      ab      "
```

## 38.11 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 default frame font.)

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.

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. This is useful for tiling small images (or image slices) without adding blank areas between the images.

If the property value is a list of the form `(height total)`, 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, the other ways to specify the line spacing are 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.

You can specify the line spacing for all lines in a frame with the **line-spacing** frame parameter (see Section 29.3.3.4 [Layout Parameters], page 534). However, if the variable **default-line-spacing** is non-**nil**, it overrides the frame’s **line-spacing** parameter. An integer value specifies the number of pixels put below lines on graphical displays. 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 value specifies the number of pixels put below lines on graphical displays. 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 overrides the default frame line spacing and the buffer local **line-spacing** variable, for the display line ending in that newline.

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.

## 38.12 Faces

A **face** is a named collection of graphical attributes: font family, foreground color, background color, optional underlining, and many others. Faces are used in Emacs to control the style of display of particular parts of the text or the frame. See section “Standard Faces” in *The GNU Emacs Manual*, for the list of faces Emacs normally comes with.

Each face has its own **face number**, which distinguishes faces at low levels within Emacs. However, for most purposes, you refer to faces in Lisp programs by the symbols that name them.

**facep object**

[Function]

This function returns **t** if *object* is a face name string or symbol (or if it is a vector of the kind used internally to record face data). It returns **nil** otherwise.

Each face name is meaningful for all frames, and by default it has the same meaning in all frames. But you can arrange to give a particular face name a special meaning in one frame if you wish.

### 38.12.1 Defining Faces

The way to define a new face is with `defface`. This creates a kind of customization item (see Chapter 14 [Customization], page 185) which the user can customize using the Customization buffer (see section “Easy Customization” in *The GNU Emacs Manual*).

`defface face spec doc [keyword value]...` [Macro]

This declares `face` as a customizable face that defaults according to `spec`. You should not quote the symbol `face`, and it should not end in ‘`-face`’ (that would be redundant). The argument `doc` specifies the face documentation. The keywords you can use in `defface` are the same as in `defgroup` and `defcustom` (see Section 14.1 [Common Keywords], page 185).

When `defface` executes, it defines the face according to `spec`, then uses any customizations that were read from the init file (see Section 39.1.2 [Init File], page 813) to override that specification.

The purpose of `spec` is to specify how the face should appear on different kinds of terminals. It should be an alist whose elements have the form (*display attrs*). Each element’s CAR, *display*, specifies a class of terminals. (The first element, if its CAR is `default`, is special—it specifies defaults for the remaining elements). The element’s CADR, *atts*, is a list of face attributes and their values; it specifies what the face should look like on that kind of terminal. The possible attributes are defined in the value of `custom-face-attributes`.

The *display* part of an element of `spec` determines which frames the element matches. If more than one element of `spec` matches a given frame, the first element that matches is the one used for that frame. There are three possibilities for *display*:

`default` This element of `spec` doesn’t match any frames; instead, it specifies defaults that apply to all frames. This kind of 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 frames. 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 frames, and the *values* are possible classifications which *display* should apply to. Here are the possible values of *characteristic*:

`type` The kind of window system the frame uses—either `graphic` (any graphics-capable display), `x`, `pc` (for the MS-DOS console), `w32` (for MS Windows 9X/NT/2K/XP), `mac` (for the Macintosh display), or `tty` (a non-graphics-capable display). See Section 38.23 [Window Systems], page 811.

<b>class</b>	What kinds of colors the frame supports—either <code>color</code> , <code>grayscale</code> , or <code>mono</code> .
<b>background</b>	The kind of background—either <code>light</code> or <code>dark</code> .
<b>min-colors</b>	An integer that represents the minimum number of colors the frame should support. This matches a frame if its <code>display-color-cells</code> value is at least the specified integer.
<b>supports</b>	Whether or not the frame can display the face attributes given in <code>value...</code> (see Section 38.12.2 [Face Attributes], page 765). See the documentation for the function <code>display-supports-face-attributes-p</code> for more information on exactly how this testing is done. See [Display Face Attribute Testing], page 556.

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 frame must match one of the `values` specified for it in `display`.

Here's how the standard face `region` is defined:

```
(defface region
  '(((class color) (min-colors 88) (background dark))
    :background "blue3")
  (((class color) (min-colors 88) (background light))
    :background "lightgoldenrod2")
  (((class color) (min-colors 16) (background dark))
    :background "blue3")
  (((class color) (min-colors 16) (background light))
    :background "lightgoldenrod2")
  (((class color) (min-colors 8))
    :background "blue" :foreground "white")
  (((type tty) (class mono))
    :inverse-video t)
  (t :background "gray"))
"Basic face for highlighting the region."
:group 'basic-faces)
```

Internally, `defface` uses the symbol property `face-defface-spec` to record the face attributes specified in `defface`, `saved-face` for the attributes saved by the user with the customization buffer, `customized-face` for the attributes customized by the user for the current session, but not saved, and `face-documentation` for the documentation string.

#### frame-background-mode

[User Option]

This option, if non-`nil`, specifies the background type to use for interpreting face definitions. If it is `dark`, then Emacs treats all frames as if they had a dark background,

regardless of their actual background colors. If it is `light`, then Emacs treats all frames as if they had a light background.

### 38.12.2 Face Attributes

The effect of using a face is determined by a fixed set of *face attributes*. This table lists all the face attributes, and what they mean. You can specify more than one face for a given piece of text; Emacs merges the attributes of all the faces to determine how to display the text. See Section 38.12.4 [Displaying Faces], page 770.

Any attribute in a face can have the value `unspecified`. This means the face doesn't specify that attribute. In face merging, when the first face fails to specify a particular attribute, that means the next face gets a chance. However, the `default` face must specify all attributes.

Some of these font attributes are meaningful only on certain kinds of displays—if your display cannot handle a certain attribute, the attribute is ignored. (The attributes `:family`, `:width`, `:height`, `:weight`, and `:slant` correspond to parts of an X Logical Font Descriptor.)

<code>:family</code>	Font family name, or fontset name (see Section 38.12.9 [Fontsets], page 774). If you specify a font family name, the wild-card characters '*' and '?' are allowed.
<code>:width</code>	Relative proportionate width, also known as the character set width or set width. This should be one of the symbols <code>ultra-condensed</code> , <code>extra-condensed</code> , <code>condensed</code> , <code>semi-condensed</code> , <code>normal</code> , <code>semi-expanded</code> , <code>expanded</code> , <code>extra-expanded</code> , or <code>ultra-expanded</code> .
<code>:height</code>	Either the font height, an integer in units of 1/10 point, a floating point number specifying the amount by which to scale the height of any underlying face, or a function, which is called with the old height (from the underlying face), and should return the new height.
<code>:weight</code>	Font weight—a symbol from this series (from most dense to most faint): <code>ultra-bold</code> , <code>extra-bold</code> , <code>bold</code> , <code>semi-bold</code> , <code>normal</code> , <code>semi-light</code> , <code>light</code> , <code>extra-light</code> , or <code>ultra-light</code> .
<code>:slant</code>	Font slant—one of the symbols <code>italic</code> , <code>oblique</code> , <code>normal</code> , <code>reverse-italic</code> , or <code>reverse-oblique</code> .
	On a text-only terminal, slanted text is displayed as half-bright, if the terminal supports the feature.
<code>:foreground</code>	Foreground color, a string. The value can be a system-defined color name, or a hexadecimal color specification of the form ' <code>#rrggbb</code> '. ('#000000' is black, '#ff0000' is red, '#00ff00' is green, '#0000ff' is blue, and '#ffffff' is white.)
<code>:background</code>	Background color, a string, like the foreground color.

**: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  $(\text{width}+7)/8$  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.

**:underline**

Whether or not characters should be underlined, and in what color. If the value is `t`, underlining uses the foreground color of the face. If the value is a string, underlining uses that color. The value `nil` means do not underline.

**:overline**

Whether or not characters should be overlined, and in what color. The value is used like that of `:underline`.

**:strike-through**

Whether or not characters should be strike-through, and in what color. The value is used like that of `:underline`.

**: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. If a list of faces is used, attributes from faces earlier in the list override those from later faces.**: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 width :color color :style style)**

This way you can explicitly specify all aspects of the box. The value `width` specifies the width of the lines to draw; it defaults to 1.

The value `color` specifies the color to draw with. The default is the foreground color of the face for simple boxes, and the background color of the face for 3D boxes.

The value *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` or omitted, a plain 2D box is used.

In older versions of Emacs, before `:family`, `:height`, `:width`, `:weight`, and `:slant` existed, these attributes were used to specify the type face. They are now semi-obsolete, but they still work:

- :font** This attribute specifies the font name.
- :bold** A non-`nil` value specifies a bold font.
- :italic** A non-`nil` value specifies an italic font.

For compatibility, you can still set these “attributes,” even though they are not real face attributes. Here is what that does:

- :font** You can specify an X font name as the “value” of this “attribute”; that sets the `:family`, `:width`, `:height`, `:weight`, and `:slant` attributes according to the font name.  
If the value is a pattern with wildcards, the first font that matches the pattern is used to set these attributes.
- :bold** A non-`nil` makes the face bold; `nil` makes it normal. This actually works by setting the `:weight` attribute.
- :italic** A non-`nil` makes the face italic; `nil` makes it normal. This actually works by setting the `:slant` attribute.

#### **x-bitmap-file-path** [Variable]

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.

### 38.12.3 Face Attribute Functions

This section describes the functions for accessing and modifying the attributes of an existing face.

#### **set-face-attribute face frame &rest arguments** [Function]

This function sets one or more attributes of face *face* for frame *frame*. The attributes you specify this way override whatever the `defface` says.

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 corresponding values. Thus,

```
(set-face-attribute 'foo nil
                   :width 'extended
                   :weight 'bold)
```

```
:underline "red")
```

sets the attributes `:width`, `:weight` and `:underline` to the corresponding values.

If `frame` is `t`, this function sets the default attributes for new frames. Default attribute values specified this way override the `defface` for newly created frames.

If `frame` is `nil`, this function sets the attributes for all existing frames, and the default for new frames.

**face-attribute** *face attribute &optional frame inherit* [Function]

This returns the value of the `attribute` attribute of face `face` on `frame`. If `frame` is `nil`, that means the selected frame (see Section 29.9 [Input Focus], page 543).

If `frame` is `t`, this returns whatever new-frames default value you previously specified with `set-face-attribute` for the `attribute` attribute of `face`. If you have not specified one, it returns `nil`.

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 values are also relative.

For example:

```
(face-attribute-relative-p :height 2.0)
  ⇒ t
```

**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.

The functions above did not exist before Emacs 21. For compatibility with older Emacs versions, you can use the following functions to set and examine the face attributes which existed in those versions. They use values of `t` and `nil` for `frame` just like `set-face-attribute` and `face-attribute`.

**set-face-foreground** *face color &optional frame* [Function]  
**set-face-background** *face color &optional frame* [Function]

These functions set the foreground (or background, respectively) color of face *face* to *color*. The argument *color* should be a string, the name of a color.

Certain shades of gray are implemented by stipple patterns on black-and-white screens.

**set-face-stipple** *face pattern &optional frame* [Function]

This function sets the background stipple pattern of face *face* to *pattern*. The argument *pattern* should be the name of a stipple pattern defined by the X server, or actual bitmap data (see Section 38.12.2 [Face Attributes], page 765), or **nil** meaning don't use stipple.

Normally there is no need to pay attention to stipple patterns, because they are used automatically to handle certain shades of gray.

**set-face-font** *face font &optional frame* [Function]

This function sets the font of face *face*. This actually sets the attributes **:family**, **:width**, **:height**, **:weight**, and **:slant** according to the font name *font*.

**set-face-bold-p** *face bold-p &optional frame* [Function]

This function specifies whether *face* should be bold. If *bold-p* is non-**nil**, that means yes; **nil** means no. This actually sets the **:weight** attribute.

**set-face-italic-p** *face italic-p &optional frame* [Function]

This function specifies whether *face* should be italic. If *italic-p* is non-**nil**, that means yes; **nil** means no. This actually sets the **:slant** attribute.

**set-face-underline-p** *face underline &optional frame* [Function]

This function sets the underline attribute of face *face*. Non-**nil** means do underline; **nil** means don't. If *underline* is a string, underline with that color.

**set-face-inverse-video-p** *face inverse-video-p &optional frame* [Function]

This function sets the **:inverse-video** attribute of face *face*.

**invert-face** *face &optional frame* [Function]

This function swaps the foreground and background colors of face *face*.

These functions examine the attributes of a face. If you don't specify *frame*, they refer to the selected frame; **t** refers to the default data for new frames. They return the symbol **unspecified** if the face doesn't define any value for that attribute.

**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 *inherit* is **nil**, only a color 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 color is found. To ensure that the return value is always specified, use a value of **default** for *inherit*.

**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.

If *inherit* is `nil`, only a stipple 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 stipple is found. To ensure that the return value is always specified, use a value of `default` for *inherit*.

**face-font** *face &optional frame* [Function]

This function returns the name of the font of face *face*.

**face-bold-p** *face &optional frame* [Function]

This function returns `t` if *face* is bold—that is, if it is bolder than normal. It returns `nil` otherwise.

**face-italic-p** *face &optional frame* [Function]

This function returns `t` if *face* is italic or oblique, `nil` otherwise.

**face-underline-p** *face &optional frame* [Function]

This function returns the `:underline` attribute of face *face*.

**face-inverse-video-p** *face &optional frame* [Function]

This function returns the `:inverse-video` attribute of face *face*.

### 38.12.4 Displaying Faces

Here are the ways to specify which faces to use for display of text:

- With defaults. The `default` face is used as the ultimate default for all text. (In Emacs 19 and 20, the `default` face is used only when no other face is specified.)
- For a mode line or header line, the face `mode-line` or `mode-line-inactive`, or `header-line`, is merged in just before `default`.
- With text properties. A character can have a `face` property; if so, the faces and face attributes specified there apply. See Section 32.19.4 [Special Properties], page 620. If the character has a `mouse-face` property, that is used instead of the `face` property when the mouse is “near enough” to the character.
- With overlays. An overlay can have `face` and `mouse-face` properties too; they apply to all the text covered by the overlay.
- With a region that is active. In Transient Mark mode, the region is highlighted with the face `region` (see section “Standard Faces” in *The GNU Emacs Manual*).
- With special glyphs. Each glyph can specify a particular face number. See Section 38.21.3 [Glyphs], page 809.

If these various sources together specify more than one face for a particular character, Emacs merges the attributes of the various faces specified. For each attribute, Emacs tries first the face of any special glyph; then the face for region highlighting, if appropriate; then the faces specified by overlays, followed by those specified by text properties, then the `mode-line` or `mode-line-inactive` or `header-line` face (if in a mode line or a header line), and last the `default` face.

When multiple overlays cover one character, an overlay with higher priority overrides those with lower priority. See Section 38.9 [Overlays], page 754.

### 38.12.5 Font Selection

Selecting a *font* means mapping the specified face attributes for a character to a font that is available on a particular display. The face attributes, as determined by face merging, specify most of the font choice, but not all. Part of the choice depends on what character it is.

If the face specifies a fontset name, that fontset determines a pattern for fonts of the given charset. If the face specifies a font family, a font pattern is constructed.

Emacs tries to find an available font for the given face attributes and character's registry and encoding. If there is a font that matches exactly, it is used, of course. The hard case is when no available font exactly fits the specification. Then Emacs looks for one that is “close”—one attribute at a time. You can specify the order to consider the attributes. In the case where a specified font family is not available, you can specify a set of mappings for alternatives to try.

**face-font-selection-order** [Variable]

This variable specifies the order of importance of the face attributes `:width`, `:height`, `:weight`, and `:slant`. The value should be a list containing those four symbols, in order of decreasing importance.

Font selection first finds the best available matches for the first attribute listed; 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.

The default is `(:width :height :weight :slant)`, which means first find the fonts closest to the specified `:width`, then—among the fonts with that width—find a best match for the specified font height, and so on.

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.

**face-font-family-alternatives** [Variable]

This variable lets you specify alternative font families to try, if a given family is specified and doesn't exist. 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-registry-alternatives** [Variable]

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, since the use of too many or too big scalable fonts can crash XFree86 servers.

#### **scalable-fonts-allowed**

[Variable]

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 '("muleindian-2$"))`

allows the use of scalable fonts with registry `muleindian-2`.

#### **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.

### 38.12.6 Functions for Working with Faces

Here are additional functions for creating and working with faces.

#### **make-face name**

[Function]

This function defines a new face named *name*, initially with all attributes `nil`. It does nothing if there is already a face named *name*.

#### **face-list**

[Function]

This function returns a list of all defined face names.

#### **copy-face old-face new-name &optional frame new-frame**

[Function]

This function defines a face named *new-name* as a copy of the existing face named *old-face*. It creates the face *new-name* if that doesn't already exist.

If the optional argument *frame* is given, this function applies only to that frame. Otherwise it applies to each frame individually, copying attributes from *old-face* in each frame to *new-face* in the same frame.

If the optional argument *new-frame* is given, then `copy-face` copies the attributes of *old-face* in *frame* to *new-name* in *new-frame*.

#### **face-id face**

[Function]

This function returns the face number of face *face*.

**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)
```

### 38.12.7 Automatic Face Assignment

This hook is used for automatically assigning faces to text in the buffer. It is part of the implementation of Font-Lock mode.

**fontification-functions** [Variable]

This variable holds a list of functions that are called by Emacs redisplay as needed to assign faces automatically to text in the buffer.

The functions are called in the order listed, with one argument, a buffer position *pos*. Each function should attempt to assign faces to the text in the current buffer starting at *pos*.

Each function should record the faces they assign by setting the `face` property. It should also add a non-`nil` `fontified` property for all the text it has assigned faces to. That property tells redisplay that faces have been assigned to that text already.

It is probably a good idea for each function 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 as they are 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.

### 38.12.8 Looking Up Fonts

**x-list-fonts pattern &optional face frame maximum** [Function]

This function returns a list of available font names that match *pattern*. If the optional arguments *face* and *frame* are specified, then the list is limited to fonts that are the same size as *face* currently is on *frame*.

The argument *pattern* should be a string, perhaps with wildcard characters: the '\*' character matches any substring, and the '?' character matches any single character. Pattern matching of font names ignores case.

If you specify *face* and *frame*, *face* should be a face name (a symbol) and *frame* should be a frame.

The optional argument *maximum* sets a limit on how many fonts to return. If this 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.

**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 29.9 [Input Focus], page 543).

The list contains a vector of the following form for each font:

```
[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.

The result list is sorted according to the current face font sort order.

**x-font-family-list** &optional *frame* [Function]

This function returns a list of the font families available for *frame*’s display. If *frame* is omitted or *nil*, it describes the selected frame’s display (see Section 29.9 [Input Focus], page 543).

The value is a list of elements of this form:

```
(family . fixed-p)
```

Here *family* is a font family, and *fixed-p* is non-*nil* if fonts of that family are fixed-pitch.

**font-list-limit** [Variable]

This variable specifies maximum number of fonts to consider in font matching. The function **x-family-fonts** will not return more than that many fonts, and font selection will consider only that many fonts when searching a matching font for face attributes. The default is currently 100.

### 38.12.9 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*

This function defines a new fontset according to the specification string *fontset-spec*. The string should have this format:

*fontpattern*, [*charsetname*:*fontname*]...

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 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,\n    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** *name character fontname &optional frame* [Function]

This function modifies the existing fontset *name* to use the font name *fontname* for the character *character*.

If *name* is *nil*, this function modifies the default fontset, whose short name is ‘**fontset-default**’.

*character* may be a cons; (*from . to*), where *from* and *to* are non-generic characters. In that case, use *fontname* for all characters in the range *from* and *to* (inclusive).

*character* may be a charset. In that case, use *fontname* for all character in the charsets.

*fontname* may be a cons; (*family . registry*), where *family* is a family name of a font (possibly including a foundry name at the head), *registry* is a registry name of a font (possibly including an encoding name at the tail).

For instance, this changes the default fontset to use a font of which registry name is ‘JISX0208.1983’ for all characters belonging to the charset `japanese-jisx0208`.

```
(set-fontset-font nil 'japanese-jisx0208 '(nil . "JISX0208.1983"))
```

**char-displayable-p** *char* [Function]

This function returns *t* if Emacs ought to be able to display *char*. 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.

## 38.13 Fringes

The *fringes* of a window are thin vertical strips down the sides that are used for displaying bitmaps that indicate truncation, continuation, horizontal scrolling, and the overlay arrow.

### 38.13.1 Fringe Size and Position

The following buffer-local variables control the position and width of the window fringes.

**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 38.15.4 [Display Margins], page 786.

**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.

The values of these variables take effect when you display the buffer in a window. If you change them while the buffer is visible, you can call **set-window-buffer** to display it once again in the same window, to make the changes take effect.

**set-window-fringes** *window left &optional right outside-margins* [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.

**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*).

### 38.13.2 Fringe Indicators

The *fringe indicators* are tiny icons Emacs displays in the window fringe (on a graphic display) to indicate truncated or continued lines, buffer boundaries, overlay arrow, 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 38.13 [Fringes], page 776. This variable is automatically buffer-local in every buffer.

**indicate-buffer-boundaries** [Variable]

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))`.

**default-indicate-buffer-boundaries** [Variable]

The value of this variable is the default value for `indicate-buffer-boundaries` in buffers that do not override it.

**fringe-indicator-alist** [Variable]

This buffer-local variable specifies the mapping from logical fringe indicators to the actual bitmaps displayed in the window fringes.

These symbols identify the logical fringe indicators:

Truncation and continuation line indicators:

`truncation, continuation.`

Buffer position indicators:

`up, down, top, bottom, top-bottom.`

Empty line indicator:

`empty-line.`

Overlay arrow indicator:

`overlay-arrow.`

Unknown bitmap indicator:

`unknown.`

The value is an alist where each element (`indicator . bitmaps`) specifies the fringe bitmaps used to display a specific logical fringe indicator.

Here, `indicator` specifies the logical indicator type, and `bitmaps` is list of symbols (`left right [left1 right1]`) which specifies the actual bitmap shown in the left or right fringe for the logical indicator.

The `left` and `right` symbols specify the bitmaps shown in the left and/or right fringe for the specific indicator. The `left1` or `right1` bitmaps are used only for the ‘bottom’ and ‘top-bottom’ indicators when the last (only) line in has no final newline. Alternatively, `bitmaps` may be a single symbol which is used in both left and right fringes.

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 (non-local) `default-fringe-indicator-alist` is used.

To completely hide a specific indicator, set the bitmap to `nil`.

**default-fringe-indicator-alist** [Variable]

The value of this variable is the default value for `fringe-indicator-alist` in buffers that do not override it.

Standard fringe bitmaps for indicators:

```
left-arrow right-arrow up-arrow down-arrow
left-curly-arrow right-curly-arrow
left-triangle right-triangle
top-left-angle top-right-angle
bottom-left-angle bottom-right-angle
left-bracket right-bracket
filled-rectangle hollow-rectangle
```

```
filled-square hollow-square
vertical-bar horizontal-bar
empty-line question-mark
```

### 38.13.3 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.

Logical cursor types:

```
box , hollow, bar, hbar, hollow-small.
```

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.

#### `overflow-newline-into-fringe`

[Variable]

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 (`cursor . bitmap`) specifies the fringe bitmaps used to display a specific logical cursor type in the fringe. Here, `cursor` specifies the logical cursor type and `bitmap` is a symbol specifying the fringe bitmap to be displayed for that logical cursor type.

When `fringe-cursor-alist` has a buffer-local value, and there is no bitmap defined for a cursor type, the corresponding value from the (non-local) `default-fringes-indicator-alist` is used.

#### `default-fringes-cursor-alist`

[Variable]

The value of this variable is the default value for `fringe-cursor-alist` in buffers that do not override it.

Standard bitmaps for displaying the cursor in right fringe:

```
filled-rectangle hollow-rectangle filled-square hollow-square
vertical-bar horizontal-bar
```

### 38.13.4 Fringe Bitmaps

The *fringe bitmaps* are the actual bitmaps which represent the logical fringe indicators for truncated or continued lines, buffer boundaries, overlay arrow, etc. Fringe bitmap symbols have their own name space. The fringe bitmaps are shared by all frames and windows. You can redefine the built-in fringe bitmaps, and you can define new fringe bitmaps.

The way to display a bitmap in the left or right fringes for a given line in a window is by specifying the `display` property for one of the characters that appears in it. Use a display specification of the form (`left-fringe bitmap [face]`) or (`right-fringe bitmap [face]`) (see Section 38.15 [Display Property], page 783). Here, `bitmap` is a symbol identifying the bitmap you want, and `face` (which is optional) is the name of the face whose colors

should be used for displaying the bitmap, instead of the default `fringe` face. `face` is automatically merged with the `fringe` face, so normally `face` need only specify the foreground color for the bitmap.

**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*.

### 38.13.5 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.

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 destroy 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.

### 38.13.6 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 38.9 [Overlays], page 754).

**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; 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 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 works to can 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 38.9.2 [Overlay Properties], page 756.

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-only terminals) or fringe bitmap (for graphical terminals) 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.

## 38.14 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). See Section 29.3.3.4 [Layout Parameters], page 534.

**frame-current-scroll-bars &optional frame**

[Function]

This function reports the scroll bar type settings 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 scroll bar.) `horizontal-type` is meant to specify the horizontal scroll bar type, but since they are not implemented, it is always `nil`.

You can enable or disable scroll bars for a particular buffer, by setting the variable `vertical-scroll-bar`. This variable automatically becomes buffer-local when set. The

possible values are `left`, `right`, `t`, which means to use the frame's default, and `nil` for no scroll bar.

You can also control this for individual windows. Call the function `set-window-scroll-bars` to specify what to do for a specific window:

**set-window-scroll-bars** *window width &optional vertical-type horizontal-type* [Function]

This function sets the width and type of scroll bars for window *window*.

*width* specifies the scroll bar width 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` and `nil`, just like the values of the `vertical-scroll-bars` frame parameter.

The argument *horizontal-type* is meant to specify whether and where to have horizontal scroll bars, but since they are not implemented, it has no effect. If *window* is `nil`, the selected window is used.

**window-scroll-bars** *&optional window* [Function]

Report the width and type of scroll bars specified for *window*. If *window* is omitted or `nil`, the selected window is used. The value is a list of the form (*width cols vertical-type horizontal-type*). The value *width* is the value that was specified for the width (which may be `nil`); *cols* is the number of columns that the scroll bar actually occupies.

*horizontal-type* is not actually meaningful.

If you don't specify these values for a window with `set-window-scroll-bars`, the buffer-local variables `scroll-bar-mode` and `scroll-bar-width` in the buffer being displayed control the window's vertical 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.

**scroll-bar-mode** [Variable]

This variable, always local in all buffers, controls whether and where to put scroll bars in windows displaying the buffer. The possible values are `nil` for no scroll bar, `left` to put a scroll bar on the left, and `right` to put a scroll bar on the right.

**window-current-scroll-bars** *&optional window* [Function]

This function reports the scroll bar type for window *window*. If *window* is omitted or `nil`, the selected window is used. 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.

**scroll-bar-width** [Variable]

This variable, always local in all buffers, specifies the width of the buffer's scroll bars, measured in pixels. A value of `nil` means to use the value specified by the frame.

## 38.15 The `display` Property

The `display` text property (or overlay property) is used to insert images into text, and also control other aspects of how text displays. The value of the `display` property should be a display specification, or a list or vector containing several display specifications.

Some kinds of `display` properties specify something to display instead of the text that has the property. In this case, “the text” means all the consecutive characters that have the same Lisp object as their `display` property; these characters are replaced as a single unit. By contrast, characters that have similar but distinct Lisp objects as their `display` properties are handled separately. Here’s a function that illustrates this point:

```
(defun foo ()
  (goto-char (point-min))
  (dotimes (i 5)
    (let ((string (concat "A")))
      (put-text-property (point) (1+ (point)) 'display string)
      (forward-char 1)
      (put-text-property (point) (1+ (point)) 'display string)
      (forward-char 1))))
```

It gives each of the first ten characters in the buffer string “A” as the `display` property, but they don’t all get the same string. The first two characters get the same string, so they together are replaced with one ‘A’. The next two characters get a second string, so they together are replaced with one ‘A’. Likewise for each following pair of characters. Thus, the ten characters appear as five A’s. This function would have the same results:

```
(defun foo ()
  (goto-char (point-min))
  (dotimes (i 5)
    (let ((string (concat "A")))
      (put-text-property (point) (2+ (point)) 'display string)
      (put-text-property (point) (1+ (point)) 'display string)
      (forward-char 2))))
```

This illustrates that what matters is the property value for each character. If two consecutive characters have the same object as the `display` property value, it’s irrelevant whether they got this property from a single call to `put-text-property` or from two different calls.

The rest of this section describes several kinds of display specifications and what they mean.

### 38.15.1 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 an integer or floating point 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 38.15.2 [Pixel Specification], page 784).

**: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 width of that character, multiplied by *factor*.

**:align-to *hpos***

Specifies that the space should be wide enough to reach *hpos*. If *hpos* is a number, it is measured in units of the normal character width. *hpos* can also be a *pixel width* specification (see Section 38.15.2 [Pixel Specification], page 784).

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 an integer or floating point 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 38.15.2 [Pixel Specification], page 784).

**: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 38.15.2 [Pixel Specification], page 784).

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.

## 38.15.2 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 | 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.

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 `image` corresponds to the width or height of the image.

The `left-fringe`, `right-fringe`, `left-margin`, `right-margin`, `scroll-bar`, and `text` elements specify to the width of the corresponding area of the window.

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.

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`’ in a header-line aligns with the first text column in the text area.

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.

The form `(+ expr ...)` adds up the value of the expressions. The form `(- expr ...)` negates or subtracts the value of the expressions.

### 38.15.3 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 38.16 [Images], page 787). 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. Integer values 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 38.15.4 [Display Margins], page 786).

**(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.

**(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.

*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.

If the text also has a **height** display specification, that does not affect the amount of raising or lowering, which is based on the faces used for the text.

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.

### 38.15.4 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.

To put text in the left or right display margin of the window, use a display specification of the form `(margin right-margin)` or `(margin left-margin)` on it. To put an image in a display margin, use that display specification along with the display specification for the image. Unfortunately, there is currently no way to make text or images in the margin mouse-sensitive.

If you put such a display specification directly on text in the buffer, the specified margin display appears *instead of* that buffer text itself. To put something in the margin *in association with* certain buffer text without preventing or altering the display of that text, put a `before-string` property on the text and put the display specification on the contents of the before-string.

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. It is buffer-local in all buffers.

**right-margin-width** [Variable]

This variable specifies the width of the right margin. It is buffer-local in all buffers.

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`.

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*. The argument *left* controls the left margin and *right* controls the right margin (default 0).

**window-margins** &**optional** *window* [Function]

This function returns the left and right margins of *window* as a cons cell of the form `(left . right)`. If *window* is `nil`, the selected window is used.

## 38.16 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 38.15 [Display Property], page 783).

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 29.23 [Display Feature Testing], page 555).

Emacs can display a number of different image formats; some of them are supported only if particular support libraries are installed on your machine. In some environments, Emacs can load image libraries on demand; if so, the variable `image-library-alist` can be used to modify the set of known names for these dynamic libraries (though it is not possible to add new image formats).

The supported image formats include XBM, XPM (this requires the libraries `libXpm` version 3.4k and `libz`), GIF (requiring `libungif` 4.1.0), PostScript, PBM, JPEG (requiring the `libjpeg` library version v6a), TIFF (requiring `libtiff` v3.4), and PNG (requiring `libpng` 1.0.2).

You specify one of these formats with an image type symbol. The image type symbols are `xbm`, `xpm`, `gif`, `postscript`, `pbm`, `jpeg`, `tiff`, and `png`.

**image-types** [Variable]

This variable contains a list of those image type symbols that are potentially supported in the current configuration. *Potentially* here means that Emacs knows about the image types, not necessarily that they can be loaded (they could depend on unavailable dynamic libraries, for example).

To know which image types are really available, use `image-type-available-p`.

**image-library-alist** [Variable]

This is an alist of image types vs external libraries needed to display them.

Each element is a list (`(image-type library...)`), where the car is a supported image format from `image-types`, and the rest are strings giving alternate filenames for the corresponding external libraries to load.

Emacs tries to load the libraries in the order they appear on the list; if none is loaded, the running session of Emacs won't support the image type. `pbm` and `xbm` don't need to be listed; they're always supported.

This variable is ignored if the image libraries are statically linked into Emacs.

**image-type-available-p type** [Function]

This function returns non-`nil` if image type `type` is available, i.e., if images of this type can be loaded and displayed in Emacs. `type` should be one of the types contained in `image-types`.

For image types whose support libraries are statically linked, this function always returns `t`; for other image types, it returns `t` if the dynamic library could be loaded, `nil` otherwise.

### 38.16.1 Image Descriptors

An image descriptor is a list of the form `(image . props)`, where `props` is a property list containing alternating keyword symbols (symbols whose names start with a colon) and their values. You can use any Lisp object as a property, but the only properties that have any special meaning are certain symbols, all of them keywords.

Every image descriptor must contain the property `:type type` to specify the format of the image. The value of `type` should be an image type symbol; for example, `xpm` for an image in XPM format.

Here is a list of other properties that are meaningful for all image types:

**:file file**

The `:file` property says to load the image from file `file`. If `file` is not an absolute file name, it is expanded in `data-directory`.

**:data data**

The `:data` property says the actual contents of the image. Each image must use either `:data` or `:file`, but not both. For most image types, the value of the `:data` property should be a string containing the image data; we recommend using a unibyte string.

Before using `:data`, look for further information in the section below describing the specific image format. For some image types, `:data` may not be supported; for some, it allows other data types; for some, `:data` alone is not enough, so you need to use other image properties along with `:data`.

**:margin margin**

The `:margin` property 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**

The `:ascent` property 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, 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**

The `:relief` property, if non-`nil`, 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.

**:conversion algorithm**

The `:conversion` property, if non-`nil`, 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

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 9 & 9 & -1 \end{pmatrix}$$

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 29.17 [Pointer Shape], page 550, 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 29.17 [Pointer Shape], page 550, 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`.

#### `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 29.9 [Input Focus], page 543).

### 38.16.2 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 `:height` and `:width`.
- A string containing the same byte sequence as an XBM file would contain. You must not specify `:height` and `:width` in this case, because omitting them is what indicates the data has the format of an XBM file. The file contents specify the height and width of the image.
- 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 `width * height` bits. In this case, you must specify `:height` and `:width`,

both to indicate that the string contains just the bits rather than a whole XBM file, and to specify the size of the image.

**:width width**

The value, *width*, specifies the width of the image, in pixels.

**:height height**

The value, *height*, specifies the height of the image, in pixels.

### 38.16.3 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.

### 38.16.4 GIF Images

For GIF images, specify image type `gif`.

**:index index**

You can use `:index` to specify one image from a GIF file that contains more than one image. This property specifies use of image number *index* from the file. If the GIF file doesn't contain an image with index *index*, the image displays as a hollow box.

### 38.16.5 PostScript Images

To use PostScript for an image, specify image type `postscript`. This works only if you have Ghostscript installed. You must always use these three properties:

**:pt-width width**

The value, *width*, specifies the width of the image measured in points (1/72 inch). *width* must be an integer.

**:pt-height height**

The value, *height*, specifies the height of the image in points (1/72 inch). *height* must be an integer.

**:bounding-box box**

The value, *box*, must be a list or vector of four integers, which specifying the bounding box of the PostScript image, analogous to the ‘`BoundingBox`’ comment found in PostScript files.

```
%>BoundingBox: 22 171 567 738
```

Displaying PostScript images from Lisp data is not currently implemented, but it may be implemented by the time you read this. See the ‘etc/NEWS’ file to make sure.

### 38.16.6 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 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.

For JPEG images, specify image type `jpeg`.

For TIFF images, specify image type `tiff`.

For PNG images, specify image type `png`.

### 38.16.7 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.

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 :heuristic-mask t)
```

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`.

**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** [Variable]

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 directory names 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 name 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 that uses a common idiom to provide compatibility with versions of Emacs that lack the variable `image-load-path`:

```
image-load-path)))))  
(mh-tool-bar-folder-buttons-init))
```

### 38.16.8 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* [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. 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 38.15 [Display Property], page 783.

**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**cols* equally sized slices.

**put-image** *image pos* &optional *string area* [Function]

This function puts *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.

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!)

**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 canonical character units (fractions of the width/height of the frame's default font). *frame* is the frame on which the image will be displayed. *frame* null or omitted means use the selected frame (see Section 29.9 [Input Focus], page 543).

**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 a floating point number, 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 38.16.9 [Image Cache], page 796).

### 38.16.9 Image Cache

Emacs stores images in an image cache when it displays them, so it can display them again more efficiently. It removes an image from the cache when it hasn't been displayed for a specified period of time.

When an image is looked up in the cache, its specification is compared with cached image specifications using `equal`. This means that all images with equal specifications share the same image in the cache.

**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.

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.

**clear-image-cache &optional frame** [Function]

This function clears the image cache. If *frame* is non-*nil*, only the cache for that frame is cleared. Otherwise all frames' caches are cleared.

### 38.17 Buttons

The *button* package defines functions for inserting and manipulating clickable (with the mouse, or via keyboard commands) buttons in Emacs buffers, such as might be used for help hyper-links, etc. Emacs uses buttons for the hyper-links in help text and the like.

A button is essentially a set of properties attached (via text properties or overlays) to a region of text in an Emacs buffer. These properties are called *button properties*.

One of these properties (`action`) is a function, which will be called when the user invokes it using the keyboard or the mouse. The invoked function may then examine the button and use its other properties as desired.

In some ways the Emacs button package duplicates functionality offered by the widget package (see section “Introduction” in *The Emacs Widget Library*), but the button package has the advantage that it is much faster, much smaller, and much simpler to use (for elisp programmers—for users, the result is about the same). The extra speed and space savings are useful mainly if you need to create many buttons in a buffer (for instance an `*Apropos*` buffer uses buttons to make entries clickable, and may contain many thousands of entries).

### 38.17.1 Button Properties

Buttons have an associated list of properties defining their appearance and behavior, and other arbitrary properties may be used for application specific purposes. Some properties that have special meaning to the button package include:

`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 of the button. When creating a button, this is usually specified using the `:type` keyword argument. See Section 38.17.2 [Button Types], page 798.

`help-echo`

A string displayed by the Emacs tool-tip help system; by default, "mouse-2, RET: Push this button".

`follow-link`

The follow-link property, defining how a MOUSE-1 click behaves on this button, See Section 32.19.10 [Links and Mouse-1], page 629.

`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.

### 38.17.2 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*. 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.

### 38.17.3 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 38.17.1 [Button Properties], page 797. In addition, the keyword argument `:type` may be used to specify a button-type from which to inherit other properties; see Section 38.17.2 [Button Types], page 798. 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 38.9 [Overlays], page 754) 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 insert a button with the label *label* at point, and returns it.

The following functions are similar, but use Emacs text properties (see Section 32.19 [Text Properties], page 615) to hold the button properties, making the button actually part of the text instead of being a property of the buffer. Buttons using text properties do not create markers into the buffer, which is important for speed when you use extremely large numbers of buttons. Both functions return the position of the start 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.

### 38.17.4 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 it). 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.

**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*.

**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*.

### 38.17.5 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 region-specific keymap. 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.

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 32.19.10 [Links and Mouse-1], page 629.

**`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 21.6.3 [Mouse Events], page 317), 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`** [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.

**`backward-button n &optional wrap display-message`** [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.

**`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.

### 38.18 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.

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 correspondance between buffer positions and nodes, move point from one node’s textual representation to another, and so forth. See Section 38.18.1 [Abstract Display Functions], page 801.

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 38.18.2 [Abstract Display Example], page 803) 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.

### 38.18.1 Abstract Display Functions

In this subsection, `ewoc` and `node` stand for the structures described above (see Section 38.18 [Abstract Display], page 800), 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, argth 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*.

### 38.18.2 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
      (let ((m (make-sparse-keymap)))
        (suppress-keymap m)
        (define-key m "i" 'colorcomp-R-less)
        (define-key m "o" 'colorcomp-R-more)
        (define-key m "k" 'colorcomp-G-less)
        (define-key m "l" 'colorcomp-G-more)
        (define-key m "," 'colorcomp-B-less)
        (define-key m "." 'colorcomp-B-more)
        (define-key m " " 'colorcomp-copy-as-kill-and-exit)
        m)))

```

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.

## 38.19 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 for the cursor to remain at 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 moves the cursor 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 sexp before point."
  (interactive)
  (let ((blink-matching-paren-distance
        (buffer-size))
        (blink-matching-paren t))
    (blink-matching-open)))
```

## 38.20 Usual Display Conventions

The usual display conventions define how to display each character code. You can override these conventions by setting up a display table (see Section 38.21 [Display Tables], page 807). Here are the usual display conventions:

- Character codes 32 through 126 map to glyph codes 32 through 126. Normally this means they display as themselves.
- Character code 9 is a horizontal tab. It displays as whitespace up to a position determined by `tab-width`.
- Character code 10 is a newline.
- All other codes in the range 0 through 31, and code 127, display in one of two ways according to the value of `ctl-arrow`. If it is `non-nil`, these codes map to sequences of two glyphs, where the first glyph is the ASCII code for ‘~’. (A display table can specify a glyph to use instead of ‘~’.) Otherwise, these codes map just like the codes in the range 128 to 255.

On MS-DOS terminals, Emacs arranges by default for the character code 127 to be mapped to the glyph code 127, which normally displays as an empty polygon. This glyph is used to display non-ASCII characters that the MS-DOS terminal doesn't support. See section “MS-DOS and MULE” in *The GNU Emacs Manual*.

- Character codes 128 through 255 map to 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 ‘\’.)
- Multibyte character codes above 256 are displayed as themselves, or as a question mark or empty box if the terminal cannot display that character.

The usual 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 behavior.

These display rules apply to carriage return (character code 13), when it appears in the buffer. But that character may not appear in the buffer where you expect it, if it was eliminated as part of end-of-line conversion (see Section 33.10.1 [Coding System Basics], page 648).

These variables affect the way certain characters are displayed on the screen. Since they change the number of columns the characters occupy, they also affect the indentation functions. These variables 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 23.4 [Mode Line Format], page 402).

**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 a backslash followed by three octal digits: ‘\001’.

**default-ctl-arrow**

[Variable]

The value of this variable is the default value for `ctl-arrow` in buffers that do not override it. See Section 11.10.3 [Default Value], page 152.

**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 32.17.5 [Indent Tabs], page 613.

## 38.21 Display Tables

You can use the *display table* feature to control how all possible character codes display on the screen. This is useful for displaying European languages that have letters not in the ASCII character set.

The display table maps each character code into a sequence of *glyphs*, each glyph being a graphic that takes up one character position on the screen. You can also define how to display each glyph on your terminal, using the *glyph table*.

Display tables affect how the mode line is displayed; if you want to force redisplay of the mode line using a new display table, call `force-mode-line-update` (see Section 23.4 [Mode Line Format], page 402).

### 38.21.1 Display Table Format

A display table is actually a char-table (see Section 6.6 [Char-Tables], page 93) with `display-table` as its subtype.

**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` or a vector of the glyphs to be output (see Section 38.21.3 [Glyphs], page 809). `nil` says to display the

character *c* according to the usual display conventions (see Section 38.20 [Usual Display], page 806).

**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 38.21.3 [Glyphs], page 809. On graphical terminals, Emacs uses arrows in the fringes to indicate truncation, so the display table has no effect.
- 1 The glyph for the end of a continued line (the default is ‘\V’). On graphical terminals, Emacs uses curved arrows in the fringes to indicate continuation, so the display table has no effect.
- 2 The glyph for indicating a character displayed as an octal character code (the default is ‘\V’).
- 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 38.7 [Selective Display], page 750.
- 5 The glyph used to draw the border between side-by-side windows (the default is ‘|’). See Section 28.2 [Splitting Windows], page 498. This takes effect only when there are no scroll bars; if scroll bars are supported and in use, a scroll bar separates the two windows.

For example, here is how to construct a display table that mimics the effect of setting `ctl-arrow` to a non-*nil* value:

```
(setq disptab (make-display-table))
(let ((i 0))
  (while (< i 32)
    (or (= i ?\t) (= i ?\n)
        (aset disptab i (vector ?^ (+ i 64))))
    (setq i (1+ i)))
  (aset disptab 127 (vector ?^ ??)))
```

**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.

### 38.21.2 Active Display Table

Each window can specify a display table, and so can each buffer. When a buffer *b* is displayed in window *w*, display uses the display table for window *w* if it has one; otherwise, the display table for buffer *b* if it has one; otherwise, the standard display table if any. The display table chosen is called the *active display table*.

**window-display-table** &optional *window* [Function]

This function returns *window*'s display table, or `nil` if *window* does not have an assigned display table. 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 in a particular buffer specifies the display table for that buffer. If it is `nil`, that means the buffer does not have an assigned display table.

**standard-display-table** [Variable]

This variable's value is the default display table, used whenever a window has no display table and neither does the buffer displayed in that window. This variable is `nil` by default.

If there is no display table to use for a particular window—that is, if the window specifies none, its buffer specifies none, and **standard-display-table** is `nil`—then Emacs uses the usual display conventions for all character codes in that window. See Section 38.20 [Usual Display], page 806.

A number of functions for changing the standard display table are defined in the library '`disp-table`'.

### 38.21.3 Glyphs

A *glyph* is a generalization of a character; it stands for an image that takes up a single character position on the screen. Normally glyphs come from vectors in the display table (see Section 38.21 [Display Tables], page 807).

A glyph is represented in Lisp as a *glyph code*. A glyph code can be *simple* or it can be defined by the *glyph table*. A simple glyph code is just a way of specifying a character and a face to output it in. See Section 38.12 [Faces], page 762.

The following functions are used to manipulate simple glyph codes:

**make-glyph-code** *char* &optional *face* [Function]

This function returns a simple glyph code representing *char* *char* with face *face*.

**glyph-char** *glyph* [Function]

This function returns the character of simple glyph code *glyph*.

**glyph-face** *glyph* [Function]

This function returns face of simple glyph code *glyph*, or `nil` if *glyph* has the default face (face-id 0).

On character terminals, you can set up a *glyph table* to define the meaning of glyph codes (represented as small integers).

**glyph-table**

[Variable]

The value of this variable is the current glyph table. It should be `nil` or a vector whose *gth* element defines glyph code *g*.

If a glyph code is greater than or equal to the length of the glyph table, that code is automatically simple. If `glyph-table` is `nil` then all glyph codes are simple.

The glyph table is used only on character terminals. On graphical displays, all glyph codes are simple.

Here are the meaningful types of elements in the glyph table:

<code>string</code>	Send the characters in <i>string</i> to the terminal to output this glyph code.
<code>code</code>	Define this glyph code as an alias for glyph code <i>code</i> created by <code>make-glyph-code</code> . You can use such an alias to define a small-numbered glyph code which specifies a character with a face.
<code>nil</code>	This glyph code is simple.

**create-glyph string**

[Function]

This function returns a newly-allocated glyph code which is set up to display by sending *string* to the terminal.

## 38.22 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 10.5.3 [Errors], page 127.)

**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-only terminals provided the terminal's Termcap entry defines the visible bell capability ('vb').

**ring-bell-function**

[Variable]

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.

## 38.23 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 variable tells Lisp programs what window system Emacs is running under. The possible values are

- x Emacs is displaying using X.
- pc Emacs is displaying using MS-DOS.
- w32 Emacs is displaying using Windows.
- mac Emacs is displaying using a Macintosh.
- nil Emacs is using a character-based terminal.

### window-setup-hook

[Variable]

This variable is a normal hook which Emacs runs after handling the initialization files. Emacs runs this hook after it has completed loading your init file, the default initialization file (if any), and the terminal-specific Lisp code, and running the hook `term-setup-hook`.

This hook is used for internal purposes: setting up communication with the window system, and creating the initial window. Users should not interfere with it.

# 39 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, and flow control.

See Section E.1 [Building Emacs], page 870, for related information. See also Chapter 38 [Display], page 739, for additional operating system status information pertaining to the terminal and the screen.

## 39.1 Starting Up Emacs

This section describes what Emacs does when it is started, and how you can customize these actions.

### 39.1.1 Summary: Sequence of Actions at Startup

The order of operations performed (in ‘`startup.el`’) by Emacs when it is started up is as follows:

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 these will be scanned in their turn. The files ‘`subdirs.el`’ are normally generated automatically by Emacs installation.
2. It sets the language environment and the terminal coding system, if requested by environment variables such as `LANG`.
3. It loads the initialization library for the window system, if you are using a window system. This library’s name is ‘`term/windowsystem-win.el`’.
4. It processes the initial options. (Some of them are handled even earlier than this.)
5. It initializes the window frame and faces, if appropriate.
6. It runs the normal hook `before-init-hook`.
7. It loads the library ‘`site-start`’ (if any), unless the option ‘`-Q`’ (or ‘`--no-site-file`’) was specified. The library’s file name is usually ‘`site-start.el`’.
8. It loads your init file (usually ‘`~/.emacs`’), unless the option ‘`-q`’ (or ‘`--no-init-file`’), ‘`-Q`’, or ‘`--batch`’ was specified on the command line. The ‘`-u`’ option can specify another user whose home directory should be used instead of ‘`~`’.
9. It loads the library ‘`default`’ (if any), unless `inhibit-default-init` is non-`nil`. (This is not done in ‘`-batch`’ mode, or if ‘`-Q`’ or ‘`-q`’ was specified on the command line.) The library’s file name is usually ‘`default.el`’.
10. It runs the normal hook `after-init-hook`.
11. It sets the major mode according to `initial-major-mode`, provided the buffer ‘`*scratch*`’ is still current and still in Fundamental mode.
12. It loads the terminal-specific Lisp file, if any, except when in batch mode or using a window system.
13. It displays the initial echo area message, unless you have suppressed that with `inhibit-startup-echo-area-message`.
14. It processes the action arguments from the command line.
15. It runs `emacs-startup-hook` and then `term-setup-hook`.

16. It calls `frame-notice-user-settings`, which modifies the parameters of the selected frame according to whatever the init files specify.
17. It runs `window-setup-hook`. See Section 38.23 [Window Systems], page 811.
18. It displays copyleft, nonwarranty, and basic use information, provided the value of `inhibit-startup-message` is `nil`, you didn't specify `--no-splash` or `'-Q'`.

**inhibit-startup-message**

[User Option]

This variable inhibits the initial startup messages (the nonwarranty, etc.). If it is `non-nil`, then the messages are not printed.

This variable exists so you can set it in your personal init file, once you are familiar with the contents of the startup message. Do not set this variable in the init file of a new user, or in a way that affects more than one user, because that would prevent new users from receiving the information they are supposed to see.

**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. 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.

### 39.1.2 The Init File, ‘.emacs’

When you start Emacs, it normally attempts to load your *init file*, a file in your home directory. Its normal name is ‘`.emacs`’, but you can also call it ‘`.emacs.el`’. Alternatively, you can use a file named ‘`init.el`’ in a subdirectory ‘`.emacs.d`’. Whichever place you use, you can also compile the file (see Chapter 16 [Byte Compilation], page 214); then the actual file loaded will be ‘`.emacs.elc`’ or ‘`init.elc`’.

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.

A site may have a *default init file*, which is the library named ‘`default.el`’. Emacs finds the ‘`default.el`’ file through the standard search path for libraries (see Section 15.1 [How Programs Do Loading], page 201). The Emacs distribution does not come with this file; sites may provide one for local customizations. If the default init file exists, it is loaded whenever you start Emacs, except in batch mode or if ‘`-q`’ (or ‘`-Q`’) is specified. 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.

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** [Variable]

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]

This variable prevents Emacs from loading the default initialization library file for your session of Emacs. If its value is non-`nil`, then the default library is not loaded. The default value is `nil`.

**before-init-hook** [Variable]

This normal hook is run, once, just before loading all the init files (the user’s init file, ‘`default.el`’, and/or ‘`site-start.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 (the user’s init file, ‘`default.el`’, and/or ‘`site-start.el`’), before loading the terminal-specific library and processing the command-line action arguments.

**emacs-startup-hook** [Variable]

This normal hook is run, once, just after handling the command line arguments, just before `term-setup-hook`.

**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.

### 39.1.3 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. Emacs finds the file in the normal manner, by searching the `load-path` directories, and trying the ‘`.elc`’ and ‘`.el`’ suffixes.

The usual function 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 `function-key-map` if the Termcap or Terminfo entry does not specify all the terminal’s function keys. See Section 39.12 [Terminal Input], page 832.

When the name of the terminal type contains a hyphen, and no library is found whose name is identical to the terminal's name, Emacs strips from the terminal's name the last hyphen 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 in the name (the latter means the terminal doesn't have any library specific to it). Thus, for example, if there are no ‘aaa-48’ and ‘aaa-30’ libraries, Emacs will try the same library ‘term/aaa.el’ for terminal types ‘aaa-48’ and ‘aaa-30-rv’. If necessary, the 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`. This feature is useful when experimenting with your own peculiar customizations.

You can also arrange to override some of the actions of the terminal-specific library by setting the variable `term-setup-hook`. This is a normal hook which Emacs runs using `run-hooks` at the end of Emacs initialization, after loading both your init file and any terminal-specific libraries. You can use this variable to define initializations for terminals that do not have their own libraries. See Section 23.1 [Hooks], page 382.

**term-file-prefix**

[Variable]

If the `term-file-prefix` 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. To do this, put the following in your init file: (`(setq term-file-prefix nil)`).

On MS-DOS, if the environment variable TERM is not set, Emacs uses ‘internal’ as the terminal type.

**term-setup-hook**

[Variable]

This variable is a normal hook that Emacs runs after loading your init file, the default initialization file (if any) and the terminal-specific Lisp file.

You can use `term-setup-hook` to override the definitions made by a terminal-specific file.

See `window-setup-hook` in Section 38.23 [Window Systems], page 811, for a related feature.

### 39.1.4 Command-Line Arguments

You can use command-line arguments to request various actions when you start Emacs. Since you do not need to start Emacs more than once per day, and will often leave your Emacs session running longer than that, command-line arguments are hardly ever used. As a practical matter, it is best to avoid making the habit of using them, since this habit would encourage you to kill and restart Emacs unnecessarily often. These options exist for two reasons: to be compatible with other editors (for invocation by other programs) and to enable shell scripts to run specific Lisp programs.

This section describes how Emacs processes command-line arguments, and how you can customize them.

**command-line**

[Function]

This function parses the command line that Emacs was called with, processes it, 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`, 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]

The value of this variable is an alist of user-defined command-line options and associated handler functions. This variable exists so you can add elements to it.

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 (not 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`. (The entire list of command-line arguments is in `command-line-args`.)

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-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 used as a file name to visit.

## 39.2 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. As a practical matter, you seldom kill Emacs—only when you are about to log out. Suspending is much more common.

### 39.2.1 Killing Emacs

Killing Emacs means ending the execution of the Emacs process. The parent process normally resumes control. The low-level primitive for killing Emacs is `kill-emacs`.

`kill-emacs &optional exit-data` [Function]

This function exits the Emacs process and kills it.

If `exit-data` is an integer, then it is used as the exit status of the Emacs process. (This is useful primarily in batch operation; see Section 39.16 [Batch Mode], page 836.)

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.

All the information in the Emacs process, aside from files that have been saved, is lost when the Emacs process is killed. Because killing Emacs inadvertently can lose a lot of work, Emacs queries for confirmation before actually terminating if you have buffers that need saving or subprocesses that are running. This is done in the function `save-buffers-kill-emacs`, the higher level function from which `kill-emacs` is usually called.

`kill-emacs-query-functions` [Variable]

After asking the standard questions, `save-buffers-kill-emacs` calls the functions in the list `kill-emacs-query-functions`, in order of appearance, with no arguments. These functions 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.

`kill-emacs-hook` [Variable]

This variable is a normal hook; once `save-buffers-kill-emacs` is finished with all file saving and confirmation, it calls `kill-emacs` which runs the functions in this hook. `kill-emacs` does not run this hook in batch mode.

`kill-emacs` may be invoked directly (that is not via `save-buffers-kill-emacs`) if the terminal is disconnected, or in similar situations where interaction with the user is not possible. Thus, if your hook needs to interact with the user, put it on `kill-emacs-query-functions`; if it needs to run regardless of how Emacs is killed, put it on `kill-emacs-hook`.

### 39.2.2 Suspending Emacs

Suspending Emacs 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`.

Some operating systems 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.

Suspension is not useful with window systems, because the Emacs job may not have a parent that can resume it again, and in any case you can give input to some other job such as a shell merely by moving to a different window. Therefore, suspending is not allowed when Emacs is using a window system (X, MS Windows, or Mac).

### **suspend-emacs &optional string**

[Function]

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.

If `string` is non-`nil`, its characters are sent to be read as terminal input by Emacs’s superior shell. 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 23.1 [Hooks], page 382.

The next redisplay after resumption will redraw the entire screen, unless the variable `no-redraw-on-reenter` is non-`nil` (see Section 38.1 [Refresh Screen], page 739).

In the following example, note that ‘`pwd`’ is not echoed after Emacs is suspended. But it is read and executed by the shell.

```
(suspend-emacs)
⇒ nil

(add-hook 'suspend-hook
          (function (lambda ()
                     (or (y-or-n-p
                           "Really suspend? ")
                         (error "Suspend canceled")))))
⇒ (lambda nil
      (or (y-or-n-p "Really suspend? ")
          (error "Suspend canceled")))
(add-hook 'suspend-resume-hook
          (function (lambda () (message "Resumed!"))))
⇒ (lambda nil (message "Resumed!"))
(suspend-emacs "pwd")
⇒ nil
----- Buffer: Minibuffer -----
Really suspend? y
----- Buffer: Minibuffer -----

----- Parent Shell -----
lewis@slug[23] % /user/lewis/manual
lewis@slug[24] % fg

----- Echo Area -----
Resumed!
```

### **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.

### 39.3 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. The convenient way to test parts of this string is with `string-match`.

**system-type** [Variable]

The value of this variable is a symbol indicating the type of operating system Emacs is operating on. Here is a table of the possible values:

**alpha-vms**

VMS on the Alpha.

**aix-v3**

AIX.

**berkeley-unix**

Berkeley BSD.

**cygwin**

Cygwin.

**dgux**

Data General DGUX operating system.

**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.)

**hpx**

Hewlett-Packard HPUX operating system.

**irix**

Silicon Graphics Irix system.

**ms-dos**

Microsoft MS-DOS “operating system.” Emacs compiled with DJGPP for MS-DOS binds `system-type` to `ms-dos` even when you run it on MS-Windows.

**next-mach**

NeXT Mach-based system.

**rtu**

Masscomp RTU, UCB universe.

**unisoft-unix**

UniSoft UniPlus.

**usg-unix-v**

AT&T System V.

`vax-vms` VAX VMS.

`windows-nt`

Microsoft windows NT. The same executable supports Windows 9X, but the value of `system-type` is `windows-nt` in either case.

`xenix` SCO Xenix 386.

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. We recommend using `system-configuration` to distinguish between different operating systems.

`system-name`

[Function]

This function returns the name of the machine you are running on.

```
(system-name)
⇒ "www.gnu.org"
```

The symbol `system-name` is a variable as well as a function. In fact, the function returns whatever value the variable `system-name` currently holds. Thus, you can set the variable `system-name` in case Emacs is confused about the name of your system. The variable is also useful for constructing frame titles (see Section 29.4 [Frame Titles], page 540).

`mail-host-address`

[Variable]

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 39.4 [User Identification], page 822. (Since this is done when Emacs starts up, the value actually used is the one saved when Emacs was dumped. See Section E.1 [Building Emacs], page 870.)

`getenv var`

[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`. If `var` returns `" "` if `var` is set but null. Within Emacs, the environment variable values are kept in the Lisp variable `process-environment`.

```
(getenv "USER")
⇒ "lewis"
```

```
lewis@slug[10] % printenv
PATH=.: /user/lewis/bin:/usr/bin:/usr/local/bin
USER=lewis
TERM=ibmapa16
SHELL=/bin/csh
HOME=/user/lewis
```

`setenv variable &optional value`

[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**, **setenv** removes *variable* from the environment. Otherwise, *value* should be a string. **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.

#### **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
⇒ ("l=/usr/stanford/lib/gnuemacs/lisp"
   "PATH=.:./user/lewis/bin:/usr/class:/nfsusr/local/bin"
   "USER=lewis"
   "TERM=ibmapa16"
   "SHELL=/bin/csh"
   "HOME=/user/lewis")
```

If **process-environment** contains “duplicate” elements that specify the same environment variable, the first of these elements specifies the variable, and the other “duplicates” are ignored.

#### **path-separator**

[Variable]

This variable holds a string which 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-DOS and MS-Windows.

#### **parse-colon-path** *path*

[Function]

This function takes a search path string such as would be the value of the PATH environment variable, and splits it at the separators, returning a list of directory names. **nil** in this list stands for “use the current directory.” Although the function’s name says “colon,” it actually uses the value of **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 from which the Emacs executable was invoked, or perhaps **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. This 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.

**load-average** &optional *use-float* [Function]

This function returns the current 1-minute, 5-minute, and 15-minute load averages, in a list.

By default, the values are integers that are 100 times the system load averages, which indicate the average number of processes trying to run. If *use-float* is non-nil, then they are returned as floating point numbers and 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)

lewis@rocky[5] % uptime
 11:55am  up 1 day, 19:37,  3 users,
load average: 1.69, 0.48, 0.36
```

**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. The value is nil if Emacs is running under a window system.

**setprv** *privilege-name* &optional *setp* *getprv* [Function]

This function sets or resets a VMS privilege. (It does not exist on other systems.) The first argument is the privilege name, as a string. The second argument, *setp*, is t or nil, indicating whether the privilege is to be turned on or off. Its default is nil. The function returns t if successful, nil otherwise.

If the third argument, *getprv*, is non-nil, setprv does not change the privilege, but returns t or nil indicating whether the privilege is currently enabled.

## 39.4 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' option was used, then Lisp packages should not load any customization files or user profile.

**user-mail-address**

[Variable]

This holds the nominal email address of the user who is using Emacs. Emacs normally sets this variable to a default value after reading your init files, but not if you have already set it. So you can set the variable to some other value in your init file if you do not want to use the default value.

**user-login-name &optional uid**

[Function]

If you don't specify *uid*, this function returns the name under which the user is logged in. If the environment variable **LOGNAME** is set, that value is used. Otherwise, if the environment variable **USER** is set, that value is used. Otherwise, the value is based on the effective UID, not the real UID.

If you specify *uid*, the value is the user name that corresponds to *uid* (which should be an integer), or **nil** if there is no such user.

```
(user-login-name)
⇒ "lewis"
```

**user-real-login-name**

[Function]

This function returns the user name corresponding to Emacs's real UID. This ignores the effective UID and ignores 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.

```
(user-full-name)
⇒ "Bil Lewis"
```

If the Emacs job's user-id does not correspond to any known user (and provided **NAME** is not set), the value 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 29.4 [Frame Titles], page 540).

**user-real-uid**

[Function]

This function returns the real UID of the user. The value may be a floating point number.

```
(user-real-uid)
⇒ 19
```

**user-uid**

[Function]

This function returns the effective UID of the user. The value may be a floating point number.

## 39.5 Time of Day

This section explains how to determine the current time and the time zone.

### **current-time-string** &optional *time-value* [Function]

This function returns the current time and date as a human-readable string. The format of the string is unvarying; the number of characters used for each part is always the same, so you can reliably use **substring** to extract pieces of it. It is wise to count the characters from the beginning of the string rather than from the end, as additional information may some day be added at the end.

The argument *time-value*, if given, specifies a time to format instead of the current time. The argument should be a list whose first two elements are integers. Thus, you can use times obtained from **current-time** (see below) and from **file-attributes** (see [Definition of file-attributes], page 448). *time-value* can also be a cons of two integers, but this is considered obsolete.

```
(current-time-string)
⇒ "Wed Oct 14 22:21:05 1987"
```

### **current-time** [Function]

This function returns the system's time value as a list of three integers: (*high low microsec*). The integers *high* and *low* combine to give the number of seconds since 0:00 January 1, 1970 UTC (Coordinated Universal Time), which is  $high * 2^{16} + low$ .

The third element, *microsec*, gives the microseconds since the start of the current second (or 0 for systems that return time with the resolution of only one second).

The first two elements can be compared with file time values such as you get with the function **file-attributes**. See [Definition of file-attributes], page 448.

### **current-time-zone** &optional *time-value* [Function]

This function returns a list describing the time zone that the user is in.

The value has the form (*offset name*). Here *offset* is an integer giving the number of seconds ahead of UTC (east of Greenwich). A negative value means west of Greenwich. The second element, *name*, is a string giving the name of the time zone. Both elements 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-value*, if given, specifies a time to analyze instead of the current time. The argument should have the same form as for **current-time-string** (see above). Thus, you can use times obtained from **current-time** (see above) and from **file-attributes**. See [Definition of file-attributes], page 448.

### **set-time-zone-rule** *tz* [Function]

This function specifies the local time zone according to *tz*. If *tz* is **nil**, that means to use an implementation-defined default time zone. If *tz* is **t**, that means to use Universal Time. Otherwise, *tz* should be a string specifying a time zone rule.

**float-time** &optional *time-value* [Function]

This function returns the current time as a floating-point number of seconds since the epoch. The argument *time-value*, if given, specifies a time to convert instead of the current time. The argument should have the same form as for `current-time-string` (see above). Thus, it accepts the output of `current-time` and `file-attributes`.

*Warning:* Since the result is floating point, it may not be exact. Do not use this function if precise time stamps are required.

## 39.6 Time Conversion

These functions convert time values (lists of two or three integers) to calendrical information and vice versa. You can get time values from the functions `current-time` (see Section 39.5 [Time of Day], page 824) and `file-attributes` (see [Definition of file-attributes], page 448).

Many operating systems are limited to time values that contain 32 bits of information; these systems typically handle only the times from 1901-12-13 20:45:52 UTC through 2038-01-19 03:14:07 UTC. However, some operating systems have larger time values, and can represent times far in the past or future.

Time conversion functions always use the Gregorian calendar, even for dates before the Gregorian calendar was introduced. Year numbers count the number of years since the year 1 B.C., and do not skip zero as traditional Gregorian years do; for example, the year number `-37` represents the Gregorian year 38 B.C.

**decode-time** &optional *time* [Function]

This function converts a time value into calendrical information. If you don't specify *time*, it decodes the current time. The return value is a list of nine elements, as follows:

```
(seconds minutes hour day month year dow dst zone)
```

Here is what the elements mean:

<i>seconds</i>	The number of seconds past the minute, as an integer between 0 and 59. On some operating systems, this is 60 for leap seconds.
<i>minutes</i>	The number of minutes past the hour, as an integer between 0 and 59.
<i>hour</i>	The hour of the day, as an integer between 0 and 23.
<i>day</i>	The day of the month, as an integer between 1 and 31.
<i>month</i>	The month of the year, as an integer between 1 and 12.
<i>year</i>	The year, an integer typically greater than 1900.
<i>dow</i>	The day of week, as an integer between 0 and 6, where 0 stands for Sunday.
<i>dst</i>	<code>t</code> if daylight saving time is effect, otherwise <code>nil</code> .
<i>zone</i>	An integer indicating the time zone, as the number of seconds east of Greenwich.

**Common Lisp Note:** Common Lisp has different meanings for *dow* and *zone*.

**encode-time** *seconds minutes hour day month year &optional zone* [Function]

This function is the inverse of **decode-time**. It converts seven items of calendrical data into a time value. For the meanings of the arguments, see the table above under **decode-time**.

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 optional argument *zone* defaults to the current time zone and its daylight saving time rules. If specified, it can be either a list (as you would get from **current-time-zone**), a string as in the TZ environment variable, t for Universal Time, or an integer (as you would get from **decode-time**). The specified zone is used without any further alteration for daylight saving time.

If you pass more than seven arguments to **encode-time**, the first six are used as *seconds* through *year*, the last argument is used as *zone*, and the arguments in between are ignored. This feature makes it possible to use the elements of a list returned by **decode-time** as the arguments to **encode-time**, like this:

```
(apply 'encode-time (decode-time ...))
```

You can perform simple date arithmetic by using out-of-range values for the *seconds*, *minutes*, *hour*, *day*, and *month* arguments; for example, day 0 means the day preceding the given month.

The operating system puts limits on the range of possible time values; if you try to encode a time that is out of range, an error results. For instance, years before 1970 do not work on some systems; on others, years as early as 1901 do work.

## 39.7 Parsing and Formatting Times

These functions convert time values (lists of two or three integers) to text in a string, and vice versa.

**date-to-time** *string* [Function]

This function parses the time-string *string* and returns the corresponding time value.

**format-time-string** *format-string &optional time universal* [Function]

This function converts *time* (or the current time, if *time* is omitted) to a string according to *format-string*. 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 has a locale-specific meaning. In the default locale (named C), it is equivalent to '%A, %B %e, %Y'.

'%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.
'%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.
'%p'	This stands for 'AM' or 'PM', as appropriate.
'%r'	This is a synonym for '%I:%M:%S %p'.
'%R'	This is a synonym for '%H:%M'.
'%S'	This stands for the seconds (00-59).
'%t'	This stands for a tab character.
'%T'	This is a synonym for '%H:%M:%S'.
'%U'	This stands for the week of the year (01-52), assuming that weeks start on Sunday.
'%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 (e.g., '-0500').

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 the middle of a '%' -sequences. If you start the field width with '0', it means to pad with zeros. If you start the field width with '\_', it means to pad with spaces.

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 between ‘%’ and one of the letters in the table above. ‘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.

If *universal* is non-*nil*, that means to describe the time as Universal Time; *nil* means describe it using what Emacs believes is the local time zone (see `current-time-zone`).

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 encodes its argument using the coding system specified by `locale-coding-system` (see Section 33.12 [Locales], page 660); after `strftime` returns the resulting string, `format-time-string` decodes the string using that same coding system.

#### `seconds-to-time seconds`

[Function]

This function converts *seconds*, a floating point number of seconds since the epoch, to a time value and returns that. To perform the inverse conversion, use `float-time`.

### 39.8 Processor Run time

#### `get-internal-run-time`

[Function]

This function returns the processor run time used by Emacs as a list of three integers: (*high* *low* *microsec*). The integers *high* and *low* combine to give the number of seconds, which is *high* \*  $2^{16}$  + *low*.

The third element, *microsec*, gives the microseconds (or 0 for systems that return time with the resolution of only one second).

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`.

### 39.9 Time Calculations

These functions perform calendrical computations using time values (the kind of list that `current-time` returns).

#### `time-less-p t1 t2`

[Function]

This returns *t* if time value *t1* is less than time value *t2*.

#### `time-subtract t1 t2`

[Function]

This returns the time difference *t1* – *t2* between two time values, in the same format as a time value.

**time-add** *t1 t2*

[Function]

This returns the sum of two time values, one of which ought to represent a time difference rather than a point in time. Here is how to add a number of seconds to a time value:

```
(time-add time (seconds-to-time seconds))
```

**time-to-days** *time*

[Function]

This function returns the number of days between the beginning of year 1 and *time*.

**time-to-day-in-year** *time*

[Function]

This returns the day number within the year corresponding to *time*.

**date-leap-year-p** *year*

[Function]

This function returns **t** if *year* is a leap year.

## 39.10 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.

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 21.10 [Quitting], page 338). 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 21.9 [Waiting], page 337). 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 calls functions that can change the match data, it should save and restore the match data. See Section 34.6.4 [Saving Match Data], page 680.

**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~~’, ‘xxAM’, ‘xx~~pm~~’, ‘xxPM’, ‘xx:xx~~am~~’, ‘xx:xxAM’, ‘xx:xx~~pm~~’, or ‘xx:xxPM’. 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`.

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).

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**

[Variable]

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 20.7 [Yes-or-No Queries], page 296.

`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.

## 39.11 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 when Emacs has been idle for *secs* seconds. The value of *secs* may be an integer or a floating point number; a value of the type returned by `current-idle-time` is also allowed.

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 39.10 [Timers], page 829).

Emacs becomes “idle” when it starts waiting for user input, 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.

`current-idle-time`

[Function]

This function returns the length of time Emacs has been idle, as a list of three integers: (*high* *low* *microsec*). The integers *high* and *low* combine to give the number of seconds of idleness, which is *high* \*  $2^{16}$  + *low*.

The third element, *microsec*, gives the microseconds since the start of the current second (or 0 for systems that return time with the resolution of only one second).

The main use of this function 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 resume-timer nil
  "Timer that 'timer-function' used to reschedule itself, or nil.")

(defun timer-function ()
  ;; If the user types a command while resume-timer
  ;; is active, the next time this function is called from
  ;; its main idle timer, deactivate resume-timer.
  (when resume-timer
    (cancel-timer resume-timer)
    ...do the work for a while...
  (when taking-a-break
    (setq resume-timer
      (run-with-idle-timer
        ;; Compute an idle time break-length
        ;; more than the current value.
        (time-add (current-idle-time)
          (seconds-to-time break-length))
      nil
      'timer-function))))
```

Some idle timer functions in user Lisp packages have a loop that does a certain amount of processing each time around, and exits when (`(input-pending-p)`) is non-nil. That 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.

To avoid these problems, don’t use that technique. Instead, write such idle timers to reschedule themselves after a brief pause, using the method in the `timer-function` example above.

## 39.12 Terminal Input

This section describes functions and variables for recording or manipulating terminal input. See Chapter 38 [Display], page 739, for related functions.

### 39.12.1 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-null, 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. If *meta* is `nil`, Emacs disregards the 8th bit; this is necessary when the terminal uses it as a parity bit. If *meta* is neither `t` nor `nil`, Emacs uses all 8 bits of input unchanged. This is good for terminals that use 8-bit character sets.

If *quit-char* is non-`nil`, it specifies the character to use for quitting. Normally this character is `C-g`. See Section 21.10 [Quitting], page 338.

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; `nil` means 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`.

### 39.12.2 Recording Input

**recent-keys** [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 100 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.)

A call to `clear-this-command-keys` (see Section 21.4 [Command Loop Info], page 312) 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 '`<...>`'.

You close the dribble file by calling this function with an argument of `nil`.

This function is normally used to record the input necessary to trigger an Emacs bug, for the sake of a bug report.

```
(open-dribble-file "~/dribble")
⇒ nil
```

See also the `open-terminalscript` function (see Section 39.13 [Terminal Output], page 834).

### 39.13 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`

[Variable]

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 38.2 [Forcing Redisplay], page 739, 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`.

#### `baud-rate`

[Function]

This obsolete function returns the value of the variable `baud-rate`.

#### `send-string-to-terminal string`

[Function]

This function sends *string* to the terminal without alteration. Control characters in *string* have terminal-dependent effects. This function operates only on text terminals.

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.

You close the termscript file by calling this function with an argument of `nil`.

See also `open-dribble-file` in Section 39.12.2 [Recording Input], page 833.

```
(open-termscript "../junk/termscript")
⇒ nil
```

## 39.14 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. Emacs version 20 and earlier did not support sound at all.

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`

[Function]

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.

## 39.15 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 29.2 [Multiple Displays], page 530.

You can specify which keysyms Emacs should use for the Meta, Alt, Hyper, and Super modifiers by setting these variables:

<code>x-alt-keysym</code>	[Variable]
<code>x-meta-keysym</code>	[Variable]
<code>x-hyper-keysym</code>	[Variable]
<code>x-super-keysym</code>	[Variable]

The name of the keysym that should stand for the Alt modifier (respectively, for 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)
```

## 39.16 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`’.

Any Lisp program output that would normally go to the echo area, either using `message`, or using `prin1`, etc., with `t` as the stream, goes instead to Emacs’s standard error descriptor when in batch mode. 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.)

`noninteractive` [Variable]

This variable is `non-nil` when Emacs is running in batch mode.

## 39.17 Session Management

Emacs supports the X Session Management Protocol for suspension and restart of applications. In the X Window System, a program called the *session manager* has the responsibility to keep track of the applications that are running. During shutdown, 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 by using a hook called `emacs-save-session-functions`. Each function in this hook is called when the session manager tells Emacs 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 that a subsequent Emacs invocation will load in order to restart the saved session.

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 "(save-excursion
(switch-to-buffer \"*scratch*\"))
(insert \"I am restored\"))")
  nil)
```

## Appendix A Emacs 21 Antinews

For those users who live backwards in time, here is information about downgrading to Emacs version 21.4. We hope you will enjoy the greater simplicity that results from the absence of many Emacs 22.1 features.

### A.1 Old Lisp Features in Emacs 21

- Many unnecessary features of redisplay have been eliminated. (The earlier major release, Emacs 20, will have a completely rewritten redisplay engine, which will be even simpler.)
  - The function `redisplay` has been removed. To update the display without delay, call `(sit-for 0)`. Since it is generally considered wasteful to update the display if there are any pending input events, no replacement for `(redisplay t)` is provided.
  - The function `force-window-update` has been removed. It shouldn't be needed, since changes in window contents are detected automatically. In case they aren't, call `redraw-display` to redraw everything.
  - Point no longer moves out from underneath invisible text at the end of each command. This allows the user to detect invisible text by moving the cursor around—if the cursor gets stuck, there is something invisible in the way. If you really want cursor motion to ignore the text, try marking it as intangible.
  - Support for image maps and image slices has been removed. Emacs was always meant for editing text, anyway.
  - The mode line now accepts all text properties, as well as `:propertize` and `:eval` forms, regardless of the `risky-local-variable` property.
  - The `line-height` and `line-spacing` properties no longer have any meaning for newline characters. Such properties wouldn't make sense, since newlines are not really characters; they just tell you where to break a line.
  - Considerable simplifications have been made to the display specification `(space . props)`, which is used for displaying a space of specified width and height. Pixel-based specifications and Lisp expressions are no longer accepted.
  - Many features associated with the fringe areas have been removed, to encourage people to concentrate on the main editing area (the fringe will be completely removed in Emacs 20.) Arbitrary bitmaps can no longer be displayed in the fringe; an overlay arrow can still be displayed, but there can only be one overlay arrow at a time (any more would be confusing.) The fringe widths cannot be adjusted, and individual windows cannot have their own fringe settings. A mouse click on the fringe no longer generates a special event.
  - Individual windows cannot have their own scroll-bar settings.
  - You can no longer use ‘`default`’ in a `defface` to specify defaults for subsequent faces.
  - The function `display-supports-face-attributes-p` has been removed. In `defface` specifications, the `supports` predicate is no longer supported.
  - The functions `merge-face-attribute` and `face-attribute-relative-p` have been removed.

- The priority of faces in a list supplied by the `:inherit` face attribute has been reversed. We like to make changes like this once in a while, to keep Emacs Lisp programmers on their toes.
- The `min-colors` face attribute, used for tailoring faces to limited-color displays, does not exist. If in doubt, use colors like “white” and “black,” which ought to be defined everywhere.
- The `tty-color-mode` frame parameter does not exist. You should just trust the terminal capabilities database.
- Several simplifications have been made to mouse support:
  - Clicking `mouse-1` won’t follow links, as that is alien to the spirit of Emacs. Therefore, the `follow-link` property doesn’t have any special meaning, and the function `mouse-on-link-p` has been removed.
  - The variable `void-text-area-pointer` has been removed, so the mouse pointer shape remains unchanged when moving between valid text areas and void text areas. The `pointer` image and text properties are no longer supported.
  - Mouse events will no longer specify the timestamp, the object clicked, equivalent buffer positions (for marginal or fringe areas), glyph coordinates, or relative pixel coordinates.
- Simplifications have also been made to the way Emacs handles keymaps and key sequences:
  - The `kbd` macro is now obsolete and is no longer documented. It isn’t that difficult to write key sequences using the string and vector representations, and we want to encourage users to learn.
  - Emacs no longer supports key remapping. You can do pretty much the same thing with `substitute-key-definition`, or by advising the relevant command.
  - The `keymap` text and overlay property is now overridden by minor mode keymaps, and will not work at the ends of text properties and overlays.
  - The functions `map-keymap`, `keymap-prompt`, and `current-active-maps` have been removed.
- Process support has been pared down to a functional minimum. The functions `call-process-shell-command` and `process-file` have been deleted. Processes no longer maintain property lists, and they won’t ask any questions when the user tries to exit Emacs (which would simply be rude.) The function `signal-process` won’t accept a process object, only the process id; determining the process id from a process object is left as an exercise to the programmer.
- Networking has also been simplified: `make-network-process` and its various associated function have all been replaced with a single easy-to-use function, `open-network-stream`, which can’t use UDP, can’t act as a server, and can’t set up non-blocking connections. Also, deleting a network process with `delete-process` won’t call the sentinel.
- Many programming shortcuts have been deleted, to provide you with the enjoyment of “rolling your own.” The macros `while-no-input`, `with-local-quit`, and `with-selected-window`, along with `dynamic-completion-table` and `lazy-completion-`

**table** no longer exist. Also, there are no built-in progress reporters; with Emacs, you can take progress for granted.

- Variable aliases are no longer supported. Aliases are for functions, not for variables.
- The variables **most-positive-fixnum** and **most-negative-fixnum** do not exist. On 32 bit machines, the most positive integer is probably 134217727, and the most negative integer is probably -134217728.
- The functions **eql** and **macroexpand-all** are no longer available. However, you can find similar functions in the **c1** package.
- The list returned by **split-string** won't include null substrings for separators at the beginning or end of a string. If you want to check for such separators, do it separately.
- The function **assoc-string** has been removed. Use **assoc-ignore-case** or **assoc-ignore-representation** (which are no longer obsolete.)
- The escape sequence '\s' is always interpreted as a super modifier, never a space.
- The variable **buffer-save-without-query** has been removed, to prevent Emacs from sneakily saving buffers. Also, the hook **before-save-hook** has been removed, so if you want something to be done before saving, advise or redefine **basic-save-buffer**.
- The variable **buffer-auto-save-file-format** has been renamed to **auto-save-file-format**, and is no longer a permanent local.
- The function **visited-file-modtime** now returns a cons, instead of a list of two integers. The primitive **set-file-times** has been eliminated.
- The function **file-remote-p** is no longer available.
- When determining the filename extension, a leading dot in a filename is no longer ignored. Thus, '.emacs' is considered to have extension 'emacs', rather than being extensionless.
- Emacs looks for special file handlers in a more efficient manner: it will choose the first matching handler in **file-name-handler-alist**, rather than trying to figure out which provides the closest match.
- The **predicate** argument for **read-file-name** has been removed, and so have the variables **read-file-name-function** and **read-file-name-completion-ignore-case**. The function **read-directory-name** has also been removed.
- The functions **all-completions** and **try-completion** will no longer accept lists of strings or hash tables (it will still accept alists, obarrays, and functions.) In addition, the function **test-completion** is no longer available.
- The 'G' interactive code character is no longer supported. Use 'F' instead.
- Arbitrary Lisp functions can no longer be recorded into **buffer-undo-list**. As a consequence, **yank-undo-function** is obsolete, and has been removed.
- Emacs will never complain about commands that accumulate too much undo information, so you no longer have to worry about binding **buffer-undo-list** to **t** for such commands (though you may want to do that anyway, to avoid taking up unnecessary memory space.)
- Atomic change groups are no longer supported.
- The list returned by (**match-data t**) no longer records the buffer as a final element.

- The function `looking-back` has been removed, so we no longer have the benefit of hindsight.
- The variable `search-spaces-regexp` does not exist. Spaces always stand for themselves in regular expression searches.
- The functions `skip-chars-forward` and `skip-chars-backward` no longer accepts character classes such as ‘`[[:alpha:]]`’. All characters are created equal.
- The `yank-handler` text property no longer has any meaning. Also, `yank-excluded-properties`, `insert-for-yank`, and `insert-buffer-substring-as-yank` have all been removed.
- The variable `char-property-alist` has been deleted. Aliases are for functions, not for properties.
- The function `get-char-property-and-overlay` has been deleted. If you want the properties at a point, find the text properties at the point; then, find the overlays at the point, and find the properties on those overlays.
- Font Lock mode only manages `face` properties; you can't use font-lock keywords to specify arbitrary text properties for it to manage. After all, it is called Font Lock mode, not Arbitrary Properties Lock mode.
- The arguments to `remove-overlays` are no longer optional.
- In `replace-match`, the replacement text now inherits properties from the surrounding text.
- The variable `mode-line-format` no longer supports the `:propertize`, `%i`, and `%I` constructs. The function `format-mode-line` has been removed.
- The functions `window-inside-edges` and `window-body-height` have been removed. You should do the relevant calculations yourself, starting with `window-width` and `window-height`.
- The functions `window-pixel-edges` and `window-inside-pixel-edges` have been removed. We prefer to think in terms of lines and columns, not pixel coordinates. (Sometime in the distant past, we will do away with graphical terminals entirely, in favor of text terminals.) For similar reasons, the functions `posn-at-point`, `posn-at-x-y`, and `window-line-height` have been removed, and `pos-visible-in-window-p` no longer worries about partially visible rows.
- The macro `save-selected-window` only saves the selected window of the selected frame, so don't try selecting windows in other frames.
- The function `minibufferp` is no longer available.
- The function `modify-all-frames-parameters` has been removed (we always suspected the name was ungrammatical, anyway.)
- The `line-spacing` variable no longer accepts float values.
- The function `tool-bar-local-item-from-menu` has been deleted. If you need to make an entry in the tool bar, you can still use `tool-bar-add-item-from-menu`, but that modifies the binding in the source keymap instead of copying it into the local keymap.
- When determining the major mode, the file name takes precedence over the interpreter magic line. The variable `magic-mode-alist`, which associates certain buffer beginnings with major modes, has been eliminated.

- The hook `after-change-major-mode-hook` is not defined, and neither are `run-mode-hooks` and `delay-mode-hooks`.
- The variable `minor-mode-list` has been removed.
- `define-derived-mode` will copy abbrevs from the parent mode's abbrev table, instead of creating a new, empty abbrev table.
- There are no “system” abbrevs. When the user saves into the abbrevs file, all abbrevs are saved.
- The Warnings facility has been removed. Just use `error`.
- Several hook variables have been renamed to flout the Emacs naming conventions. We feel that consistency is boring, and having non-standard hook names encourages users to check the documentation before using a hook. For instance, the normal hook `find-file-hook` has been renamed to `find-file-hooks`, and the abnormal hook `delete-frame-functions` has been renamed to `delete-frame-hook`.
- The function `symbol-file` does not exist. If you want to know which file defined a function or variable, try grepping for it.
- The variable `load-history` records function definitions just like variable definitions, instead of indicating which functions were previously autoloaded.
- There is a new variable, `recursive-load-depth-limit`, which specifies how many times files can recursively load themselves; it is 50 by default, and `nil` means infinity. Previously, Emacs signaled an error after just 3 recursive loads, which was boring.
- Byte-compiler warnings and error messages will leave out the line and character positions, in order to exercise your debugging skills. Also, there is no `with-no-warnings` macro—instead of suppressing compiler warnings, fix your code to avoid them!
- The function `unsafep` has been removed.
- File local variables can now specify a string with text properties. Since arbitrary Lisp expressions can be embedded in text properties, this can provide you with a great deal of flexibility and power. On the other hand, `safe-local-eval-forms` and the `safe-local-eval-function` function property have no special meaning.
- You can no longer use `char-displayable-p` to test if Emacs can display a certain character.
- The function `string-to-multibyte` is no longer available.
- The `translation-table-for-input` translation table has been removed. Also, translation hash tables are no longer available, so we don't need the functions `lookup-character` and `lookup-integer`.
- The `table` argument to `translate-region` can no longer be a char-table; it has to be a string.
- The variable `auto-coding-functions` and the two functions `merge-coding-systems` and `decode-coding-inserted-region` have been deleted. The coding system property `mime-text-unsuitable` no longer has any special meaning.
- If pure storage overflows while dumping, Emacs won't tell you how much additional pure storage it needs. Try adding in increments of 20000, until you have enough.
- The variables `gc-elapsed`, `gcs-done`, and `post-gc-hook` have been garbage-collected.

# Appendix B GNU Free Documentation License

Version 1.2, November 2002

Copyright (C) 2000,2001,2002 Free Software Foundation, Inc.  
51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA

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.

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 for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

## 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 <http://www.gnu.org/copyleft/>.

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.

## **ADDEDUM: 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 1.2  
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 2, June 1991

Copyright © 1989, 1991 Free Software Foundation, Inc.  
51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA

Everyone is permitted to copy and distribute verbatim copies  
of this license document, but changing it is not allowed.

## Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Lesser General Public License instead.) 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 this service 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 make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. 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.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

## Terms and Conditions for Copying, Distribution and Modification

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The “Program”, below, refers to any such program or work, and a “work based on the Program” means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term “modification”.) Each licensee is addressed as “you”.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute 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 and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.  
You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.
2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:
  - a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
  - b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
  - c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:
  - a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
  - b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
  - c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.
5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.
7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), 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 distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
9. The Free Software Foundation may publish revised and/or new versions of the 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 a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software

which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

## **NO WARRANTY**

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, 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.
12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE 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.

## **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 convey 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) yyyy 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 2  
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, write to the Free Software  
Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,  
MA 02110-1301, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) yyyy name of author  
Gnomovision 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, the commands you use may be called something other than ‘show w’ and ‘show c’; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a “copyright disclaimer” for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright  
interest in the program ‘Gnomovision’  
(which makes passes at compilers) written  
by James Hacker.

*signature of Ty Coon, 1 April 1989*  
Ty Coon, President of Vice

This 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.

## 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.

### D.1 Emacs Lisp Coding Conventions

Here are conventions that you should follow when writing Emacs Lisp code intended for widespread use:

- Simply loading the 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.

- Since all global variables share the same name space, and all functions share another name space, you should choose a short word to distinguish your program from other Lisp programs<sup>1</sup>. Then take care to begin the names of all global variables, constants, and functions in your program with the chosen prefix. This helps avoid name conflicts.

Occasionally, for a command name intended for users to use, it is more convenient if some words come before the package's name prefix. And constructs that define functions, variables, etc., work better if they start with ‘`defun`’ or ‘`defvar`’, 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.

If one prefix is insufficient, your package can use two or three alternative common prefixes, so long as they make sense.

Separate the prefix from the rest of the symbol name with a hyphen, ‘-’. This will be consistent with Emacs itself and with most Emacs Lisp programs.

- Put a call to `provide` at the end of each separate Lisp file.
- 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.

---

<sup>1</sup> The benefits of a Common Lisp-style package system are considered not to outweigh the costs.

- If one file *foo* uses a macro defined in another file *bar*, *foo* should contain this expression before the first use of the macro:

```
(eval-when-compile (require 'bar))
```

(And the library *bar* should contain `(provide 'bar)`, to make the `require` work.) This will cause *bar* to be loaded when you byte-compile *foo*. Otherwise, you risk compiling *foo* without the necessary macro loaded, and that would produce compiled code that won't work right. See Section 13.3 [Compiling Macros], page 177.

Using `eval-when-compile` avoids loading *bar* when the compiled version of *foo* is *used*.

- Please don't require the `c1` package of Common Lisp extensions at run time. Use of this package is optional, and it is not part of the standard Emacs namespace. If your package loads `c1` at run time, that could cause name clashes for users who don't use that package.

However, there is no problem with using the `c1` package at compile time, with `(eval-when-compile (require 'c1))`. That's sufficient for using the macros in the `c1` package, because the compiler expands them before generating the byte-code.

- When defining a major mode, please follow the major mode conventions. See Section 23.2.2 [Major Mode Conventions], page 385.
- When defining a minor mode, please follow the minor mode conventions. See Section 23.3.1 [Minor Mode Conventions], page 397.
- 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`'. If the name is one word, add just '`p`'; if the name is multiple words, add '`-p`'. Examples are `framep` and `frame-live-p`.
- If a user option variable records a true-or-false condition, give it a name that ends in '`-flag`'.
- 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 23.1 [Hooks], page 382.
- If loading the file adds functions to hooks, define a function `feature-unload-hook`, where `feature` is the name of the feature the package provides, and make it undo any such changes. Using `unload-feature` to unload the file will run this function. See Section 15.9 [Unloading], page 211.
- 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. In any case, it is a problem for debugging, because the advised function doesn't do what its source code says it does. If the programmer investigating the problem is unaware that there is advice on the function, the experience can be very frustrating. We hope to remove all the places in Emacs that advise primitives. In the mean time, please don't add any more.
- It is likewise a bad idea for one Lisp package to advise a function in another Lisp package.
- Likewise, avoid using `eval-after-load` (see Section 15.10 [Hooks for Loading], page 212) in libraries and packages. 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 functions or library programs of standard 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 '`def`'.
- A macro that defines a function or variable should have a name that starts 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.
- Please keep the names of your Emacs Lisp source files to 13 characters or less. This way, if the files are compiled, the compiled files' names will be 14 characters or less, which is short enough to fit on all kinds of Unix systems.
- 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.) The users will find Emacs more coherent if all libraries use the same conventions.
- If your program contains non-ASCII characters in string or character constants, you should make sure Emacs always decodes these characters the same way, regardless of the user's settings. There are two ways to do that:
  - Use coding system `emacs-mule`, and specify that for `coding` in the '`-*-`' line or the local variables list.
 

```
;; XXX.el -*- coding: emacs-mule; -*-
```
  - Use one of the coding systems based on ISO 2022 (such as `iso-8859-n` and `iso-2022-7bit`), and specify it with '`!`' at the end for `coding`. (The '`!`' turns off any possible character translation.)
 

```
;; XXX.el -*- coding: iso-latin-2!; -*-
```
- Indent each function with `C-M-q` (`indent-sexp`) using the default indentation parameters.
- Don't make a habit of putting close-parentheses on lines by themselves; Lisp programmers find this disconcerting. Once in a while, when there is a sequence of many

consecutive close-parentheses, it may make sense to split the sequence in one or two significant places.

- Please put a copyright notice and copying permission notice on the file if you distribute copies. Use a notice like this one:

```
;; Copyright (C) year name
;;
;; 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 2 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, write to the Free
;; Software Foundation, Inc., 51 Franklin Street, Fifth Floor,
;; Boston, MA 02110-1301 USA
```

If you have signed papers to assign the copyright to the Foundation, then use ‘Free Software Foundation, Inc.’ as name. Otherwise, use your name. See also See Section D.8 [Library Headers], page 867.

## D.2 Key Binding Conventions

- Special major modes used for read-only text should usually redefine `mouse-2` and RET to trace some sort of reference in the text. Modes such as Dired, Info, Compilation, and Occur redefine it in this way.

In addition, they should mark the text as a kind of “link” so that `mouse-1` will follow it also. See Section 32.19.10 [Links and Mouse-1], page 629.

- Please do not define `C-c letter` as a key in Lisp programs. Sequences consisting of `C-c` and a letter (either upper or lower case) 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.
- Applications should not bind mouse events based on button 1 with the shift key held down. These events include `S-mouse-1`, `M-S-mouse-1`, `C-S-mouse-1`, and so on. They are reserved for users.
- 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 punctuation 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.

- Do not 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.
  - Do not 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.
- Anything which acts like a temporary mode or state which the user can enter and leave should define `ESC ESC` or `ESC ESC ESC` as a way to escape.

For a state which 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 30.2.4 [Text Lines], page 562.
- 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 31.7 [The Mark], page 577.

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 38.4 [The Echo Area], page 741.
- When you encounter an error condition, call the function `error` (or `signal`). The function `error` does not return. See Section 10.5.3.1 [Signaling Errors], page 128.

Do not 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.
- A question asked in the minibuffer with `y-or-n-p` or `yes-or-no-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`’.
- Try to avoid using recursive edits. Instead, do what the Rmail `e` command does: use a new local keymap that contains one command defined to switch back to the old local keymap. Or do what the `edit-options` command does: switch to another buffer and let the user switch back at will. See Section 21.12 [Recursive Editing], page 342.

## D.4 Tips for Making Compiled Code Fast

Here are ways of improving the execution speed of byte-compiled Lisp programs.

- Profile your program with the ‘`elp`’ library. See the file ‘`elp.el`’ for instructions.
- Check the speed of individual Emacs Lisp forms using the ‘`benchmark`’ library. See the functions `benchmark-run` and `benchmark-run-compiled` in ‘`benchmark.el`’.
- 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 90):

```
(get 'aref 'byte-compile)
⇒ byte-compile-two-args
```

- 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 12.10 [Inline Functions], page 173.

## 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.

- If you use many functions and variables from a certain file, you can add a `require` for that package to avoid compilation warnings for them. 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 usually is a mistake but it’s not a mistake in this one case, is to put a call to `with-no-warnings` around it.

## 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. In earlier Emacs versions, you could save space by using a comment instead of a documentation string, but that is no longer the case—documentation strings now 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. However, rather than blindly filling the entire documentation string, you can often make it much more readable by choosing certain line breaks with care. Use blank lines between topics 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 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?”

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, 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 second 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* which 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 a documentation string refers to a Lisp symbol, write it as it would be printed (which usually means in lower case), with single-quotes around it. For example: ‘`lambda`’. There are two exceptions: write `t` and `nil` without single-quotes.

Help mode automatically creates a hyperlink when a documentation string uses a symbol name inside single quotes, 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 name of the Info node (or anchor) in single quotes, 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’.

Finally, to create a hyperlink to URLs, write the URL in single quotes, preceded by ‘`URL`’. For example,

The home page for the GNU project has more information (see URL  
‘<http://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 24.3 [Keys in Documentation], page 428.

- 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 ‘`\\\[...]`’. The text inside the ‘`\\\<...>`’ should be the name of the variable containing the local keymap for the major mode.

It is not practical to use ‘`\\\[...]`’ very many times, because display of the documentation string will become slow. So use this to describe the most important commands in your major mode, and then use ‘`\\\{...}`’ to display the rest of the mode’s keymap.

- 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.
- If a line in a documentation string begins with an open-parenthesis, write 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 prevents the open-parenthesis from being treated as the start of a defun (see section “Defuns” in *The GNU Emacs Manual*).

- 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.”
- 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 users ought to set interactively, you normally should use `defcustom`. However, if for some reason you use `defvar` instead, start the doc string with a ‘\*’. See Section 11.5 [Defining Variables], page 139.
- 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.

## D.7 Tips on Writing Comments

We recommend these conventions for where to put comments and how to indent them:

' ;'

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 the same line does its job. In Lisp mode and related modes, the *M-;* (*indent-for-comment*) command automatically inserts such a ‘ ;’ in the right place, or aligns such a comment if it is already present.

This and following examples are taken from the Emacs sources.

```
(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.
```

Every function that has no documentation string (presumably one that is used only internally within the package it belongs to), 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.

' ; ; ;'

Comments that start with three semicolons, ‘ ; ; ;’, should start at the left margin. These are used, occasionally, for comments within functions that should start at the margin. We also use them sometimes for comments that are between functions—whether to use two or three semicolons depends on whether the comment 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 headings, comments starting with two or less are not.

Another use for triple-semicolon comments is for commenting out lines within a function. We use three semicolons for this precisely so that they remain at the left margin. By default, Outline minor mode does not consider a comment to be a heading (even if it starts with at least three semicolons) if the semicolons are followed by at least two spaces. Thus, if you add an introductory comment to the commented out code, make sure to indent it by at least two spaces after the three semicolons.

```
(defun foo (a)
  ; ; This is no longer necessary.
  ; ; (force-mode-line-update)
  (message "Finished with %s" a))
```

When commenting out entire functions, use two semicolons.

‘;;;;’      Comments that start with four semicolons, ‘;;;;’, should be aligned to the left margin and are used for headings of major sections of a program. For example:

;;;; The kill ring

The indentation commands of the Lisp modes in Emacs, such as *M-;* (*indent-for-comment*) and TAB (*lisp-indent-line*), automatically indent comments according to these conventions, 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. This section explains these conventions.

We’ll start with an example, a package that is included in the Emacs distribution.

Parts of this example reflect its status as part of Emacs; for example, the copyright notice lists the Free Software Foundation as the copyright holder, and the copying permission says the file is part of Emacs. When you write a package and post it, the copyright holder would be you (unless your employer claims to own it instead), and you should get the suggested copying permission from the end of the GNU General Public License itself. Don’t say your file is part of Emacs if we haven’t installed it in Emacs yet!

With that warning out of the way, on to the example:

```
;; lisp-mnt.el --- minor mode for Emacs Lisp maintainers

;; Copyright (C) 1992 Free Software Foundation, Inc.

;; Author: Eric S. Raymond <esr@snark.thyrsus.com>
;; Maintainer: Eric S. Raymond <esr@snark.thyrsus.com>
;; Created: 14 Jul 1992
;; Version: 1.2
;; Keywords: docs

;; This file is part of GNU Emacs.

...
;; Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor,
;; Boston, MA 02110-1301, USA.
```

The very first line should have this format:

;; *filename* --- *description*

The description should be complete in one line. If the file needs a ‘-\*-’ specification, put it after *description*.

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 line states the name and net address of at least the principal author of the library.

If there are multiple authors, you can list them on continuation lines led by ;; and a tab character, like this:

```
;; Author: Ashwin Ram <Ram-Ashwin@cs.yale.edu>
;;       Dave Sill <de5@ornl.gov>
;;       Dave Brennan <brennan@hal.com>
;;       Eric Raymond <esr@snark.thyrsus.com>
```

**'Maintainer'**

This line should contain a single name/address as in the Author line, or an address only, or the string ‘FSF’. If there is no maintainer line, the person(s) in the Author field are presumed to be the maintainers. The example above is mildly bogus because the maintainer line is redundant.

The idea behind the ‘Author’ and ‘Maintainer’ lines is to make possible a Lisp function to “send mail to the maintainer” without having to mine the name out by hand.

Be sure to surround the network address with ‘<...>’ if you include the person’s full name as well as the network address.

**'Created'** This optional line gives the original creation date of the file. For historical interest only.

**'Version'** If you wish to record version numbers for the individual Lisp program, put them in this line.

**'Adapted-By'**

In this header line, place the name of the person who adapted the library for installation (to make it fit the style conventions, for example).

**'Keywords'**

This line lists keywords for the `finder-by-keyword` help command. Please use that command to see a list of the meaningful keywords.

This field is important; it’s how people will find your package when they’re looking for things by topic area. To separate the keywords, you can use spaces, commas, or both.

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 by blank lines from anything else. 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.

**';;; Documentation:'**

This was used in some files in place of ‘;;; Commentary:’, but it is deprecated.

**';;; Change Log:'**

This begins change log information stored in the library file (if you store the change history there). For Lisp files distributed with Emacs, the change history is kept in the file ‘`ChangeLog`’ and not in the source file at all; these files generally do not have a ‘;;; Change Log:’ line. ‘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 maintenance.

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`’ uses ‘`temacs`’ to create the real runnable Emacs executable. These arguments direct ‘`temacs`’ to evaluate the Lisp files specified in the file ‘`loadup.el`’. These files set up the normal Emacs editing environment, resulting in an Emacs that is still impure but no longer bare.

It takes a substantial time to load the standard Lisp files. Luckily, you don't have to do this each time you run Emacs; ‘`temacs`’ can dump out an executable program called ‘`emacs`’ that has these files preloaded. ‘`emacs`’ starts more quickly because it does not need to load the files. This is the Emacs executable that is normally installed.

To create ‘`emacs`’, use the command ‘`temacs -batch -l loadup dump`’. The purpose of ‘`-batch`’ here is to prevent ‘`temacs`’ from trying to initialize any of its data on the terminal; this ensures that the tables of terminal information are empty in the dumped Emacs. The argument ‘`dump`’ tells ‘`loadup.el`’ to dump a new executable named ‘`emacs`’.

Some operating systems don't support dumping. On those systems, you must start Emacs with the ‘`temacs -l loadup`’ command each time you use it. This takes a substantial time, but since you need to start Emacs once a day at most—or once a week if you never log out—the extra time is not too severe a problem.

You can specify additional files to preload by writing a library named ‘`site-load.el`’ that loads them. You may need to add a definition

```
#define SITELOAD_PURESIZE_EXTRA n
```

to make *n* added bytes of pure space to hold the additional files. (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 428).

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 non-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 39.1.1 [Startup Summary], page 812.

In a package that can be preloaded, it is sometimes useful to specify a computation to be done when Emacs subsequently starts up. For this, use `eval-at-startup`:

`eval-at-startup body...`

[Macro]

This evaluates the *body* forms, either immediately if running in an Emacs that has already started up, or later when Emacs does start up. Since the value of the *body* forms is not necessarily available when the `eval-at-startup` form is run, that form always returns `nil`.

`dump-emacs to-file from-file`

[Function]

This function dumps the current state of Emacs into an executable file *to-file*. It takes symbols from *from-file* (this is normally the executable file ‘`temacs`’).

If you want to use this function in an Emacs that was already dumped, you must run Emacs with ‘`-batch`’.

## 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 the following section for information on normal storage. 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 that happens, you should increase the compilation parameter `PURESIZE` in the file ‘`src/puresize.h`’ and rebuild Emacs, even though the resulting image will work: garbage collection 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.

**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 the file ‘`emacs/lisp/loaddefs.el`’, but a few packages call it just in case you decide to preload them.

**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 sharable 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 in blocks of 1k bytes. Each block is used for one type of Lisp object, so symbols, cons cells, markers, etc., are segregated in distinct blocks in memory. (Vectors, long strings, buffers and certain other editing types, which are fairly large, are allocated in individual blocks, one per object, while small strings are packed into blocks of 8k bytes.)

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. (This name is traditional, but “garbage recycler” might be a more intuitive metaphor for this facility.)

The garbage collector operates 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.

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.

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. Vectors, buffers, windows, and other large objects are individually allocated and freed using `malloc` and `free`.

**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 containing the following information:

```
((used-conses . free-conses)
 (used-syms . free-syms)
 (used-miscs . free-miscs)
 used-string-chars
 used-vector-slots
 (used-floats . free-floats)
 (used-intervals . free-intervals)
 (used-strings . free-strings))
```

Here is an example:

```
(garbage-collect)
⇒ ((106886 . 13184) (9769 . 0)
     (7731 . 4651) 347543 121628
     (31 . 94) (1273 . 168)
     (25474 . 3569))
```

Here is a table explaining each element:

#### `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.

#### `used-syms`

The number of symbols in use.

#### `free-syms`

The number of symbols for which space has been obtained from the operating system, but that are not currently being used.

#### `used-miscs`

The number of miscellaneous objects in use. These include markers and overlays, plus certain objects not visible to users.

*free-miscs* The number of miscellaneous objects for which space has been obtained from the operating system, but that are not currently being used.

*used-string-chars*

The total size of all strings, in characters.

*used-vector-slots*

The total number of elements of existing vectors.

*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.

*used-intervals*

The number of intervals in use. Intervals are an internal data structure used for representing text properties.

*free-intervals*

The number of intervals for which space has been obtained from the operating system, but that are not currently being used.

*used-strings*

The number of strings in use.

*free-strings*

The number of string headers for which the space was obtained from the operating system, but which are currently not in use. (A string object consists of a header and the storage for the string text itself; the latter is only allocated when the string is created.)

If there was overflow in pure space (see the previous section), `garbage-collect` returns `nil`, because a real garbage collection can not be done in this situation.

**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`, meaning there are no such messages.

**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. A cons cell counts as eight bytes, a string as one byte per character plus a few bytes of overhead, and so on; space allocated to the contents of buffers does not count. Note that the subsequent garbage collection does not happen immediately when the threshold is exhausted, but only the next time the Lisp evaluator is called.

The initial threshold value is 400,000. If you specify a larger value, garbage collection will happen less often. This reduces the amount of time spent garbage collecting, but

increases total memory use. You may want to do this when running a program that creates lots of Lisp data.

You can make collections more frequent by specifying a smaller value, down to 10,000. A value less than 10,000 will remain in effect only until the subsequent garbage collection, at which time `garbage-collect` will set the threshold back to 10,000.

**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.

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 the address of the last byte Emacs has allocated, divided by 1024. We divide the value by 1024 to make sure it fits in a Lisp integer.

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 close to 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.

**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.

## E.4 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.

**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 Emacs session.

**misc-objects-consed** [Variable]

The total number of miscellaneous objects that have been allocated so far in this Emacs session. These include markers and overlays, plus certain objects not visible to users.

**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.5 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 ...) */
  (args)
    Lisp_Object args;
{
  register Lisp_Object val = Qnil;
  struct gcpro gcpro1;

  GCPRO1 (args);

  while (CONSP (args))
  {
    val = Feval (XCAR (args));
    if (!NILP (val))
      break;
    args = XCDR (args);
  }
}
```

```

UNGC PRO;
return val;
}

```

Let's start with a precise explanation of the arguments to the DEFUN macro. Here is a template for them:

<code>DEFUN (lname, fname, sname, min, max, interactive, doc)</code>	
<code>lname</code>	This is the name of the Lisp symbol to define as the function name; in the example above, it is <code>or</code> .
<code>fname</code>	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 <code>For</code> . Remember that the arguments must be of type <code>Lisp_Object</code> ; various macros and functions for creating values of type <code>Lisp_Object</code> are declared in the file ' <code>lisp.h</code> '.
<code>sname</code>	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 <code>fname</code> with 'F' replaced with 'S'.
<code>min</code>	This is the minimum number of arguments that the function requires. The function <code>or</code> allows a minimum of zero arguments.
<code>max</code>	This is the maximum number of arguments that the function accepts, if there is a fixed maximum. Alternatively, it can be <code>UNEVALLED</code> , indicating a special form that receives unevaluated arguments, or <code>MANY</code> , indicating an unlimited number of evaluated arguments (the equivalent of <code>&amp;rest</code> ). Both <code>UNEVALLED</code> and <code>MANY</code> are macros. If <code>max</code> is a number, it may not be less than <code>min</code> and it may not be greater than eight.
<code>interactive</code>	This is an interactive specification, a string such as might be used as the argument of <code>interactive</code> in a Lisp function. In the case of <code>or</code> , it is <code>0</code> (a null pointer), indicating that <code>or</code> cannot be called interactively. A value of <code>" "</code> indicates a function that should receive no arguments when called interactively.
<code>doc</code>	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 ' <code>doc:</code> ' identifies the comment that follows as the documentation string. The ' <code>/*</code> ' and ' <code>*/</code> ' 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 ' <code>usage:</code> ', 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. ' <code>usage:</code> ' is required if the function has an unlimited number of arguments.  All the usual rules for documentation strings in Lisp code (see Section D.6 [Documentation Tips], page 862) apply to C code documentation strings too.

After the call to the `DEFUN` macro, you must write the argument name list that every C function must have, followed by ordinary C declarations for the arguments. For a function with a fixed maximum number of arguments, declare a C argument for each Lisp argument, and give them all type `Lisp_Object`. When a Lisp function has no upper limit on the number of arguments, its implementation in C actually receives exactly two arguments: the first is the number of Lisp arguments, and the second is the address of a block containing their values. They have types `int` and `Lisp_Object *`.

Within the function `For` itself, note the use of the macros `GCPRO1` and `UNGCPRO`. `GCPRO1` is used to “protect” a variable from garbage collection—to inform the garbage collector that it must look in that variable and regard its contents as an accessible object. GC protection is necessary whenever you call `Eval` or anything that can directly or indirectly call `Eval`. At such a time, any Lisp object that this function may refer to again must be protected somehow.

It suffices to ensure that at least one pointer to each object is GC-protected; that way, the object cannot be recycled, so all pointers to it remain valid. Thus, a particular local variable can do without protection if it is certain that the object it points to will be preserved by some other pointer (such as another local variable which has a `GCPRO`)<sup>1</sup>. Otherwise, the local variable needs a `GCPRO`.

The macro `GCPRO1` protects just one local variable. If you want to protect two variables, use `GCPRO2` instead; repeating `GCPRO1` will not work. Macros `GCPRO3`, `GCPRO4`, `GCPRO5`, and `GCPRO6` also exist. All these macros implicitly use local variables such as `gcprom`; you must declare these explicitly, with type `struct gcprom`. Thus, if you use `GCPRO2`, you must declare `gcprom1` and `gcprom2`. Alas, we can’t explain all the tricky details here.

`UNGCPRO` cancels the protection of the variables that are protected in the current function. It is necessary to do this explicitly.

Built-in functions that take a variable number of arguments actually accept two arguments at the C level: the number of Lisp arguments, and a `Lisp_Object *` pointer to a C vector containing those Lisp arguments. This C vector may be part of a Lisp vector, but it need not be. The responsibility for using `GCPRO` to protect the Lisp arguments from GC if necessary rests with the caller in this case, since the caller allocated or found the storage for them.

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 871.

Do not use static variables within functions—place all static variables at top level in the file. This is necessary because Emacs on some operating systems defines the keyword `static` as a null macro. (This definition is used because those systems put all variables declared static in a place that becomes read-only after dumping, whether they have initializers or not.)

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:

---

<sup>1</sup> Formerly, strings were a special exception; in older Emacs versions, every local variable that might point to a string needed a `GCPRO`.

```
defsubr (&subr-structure-name);
```

Here *subr-structure-name* 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.

If you 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,
"xSpecify coordinate pair: \nXExpression which evals to window: ",
"Return non-nil if COORDINATES is in WINDOW.\n\
COORDINATES is a cons of the form (X . Y), X and Y being distances\n\
...
If they are on the border between WINDOW and its right sibling,\n\
'vertical-line' is returned.")
(coordinates, window)
register Lisp_Object coordinates, window;
{
    int x, y;

    CHECK_LIVE_WINDOW (window, 0);
    CHECK_CONS (coordinates, 1);
    x = XINT (Fcadr (coordinates));
    y = XINT (Fcddr (coordinates));

    switch (coordinates_in_window (XWINDOW (window), &x, &y))
    {
        case 0: /* NOT in window at all. */
            return Qnil;

        case 1: /* In text part of window. */
            return Fcons (make_number (x), make_number (y));

        case 2: /* In mode line of window. */
            return Qmode_line;

        case 3: /* On right border of window. */
            return Qvertical_line;
```

```
    default:  
        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. Since `Ffuncall` can call the evaluator, you must protect pointers from garbage collection around the call to `Ffuncall`.

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, update the code in ‘byte-opt.el’ which binds `side-effect-free-fns` and `side-effect-and-error-free-fns` so that the compiler optimizer knows about it.

## E.6 Object Internals

GNU Emacs Lisp manipulates many different types of data. The actual data are stored in a heap and the only access that programs have to it is through pointers. Pointers are thirty-two bits wide in most implementations. Depending on the operating system and type of machine for which you compile Emacs, twenty-nine bits are used to address the object, and the remaining three bits are used for the tag that identifies the object's type.

Because Lisp objects are represented as tagged pointers, it is always possible to determine the Lisp data type of any object. The C data type `Lisp_Object` can hold any Lisp object of any data type. Ordinary variables have type `Lisp_Object`, which means they can hold any type of Lisp value; you can determine the actual data type only at run time. The same is true for function arguments; if you want a function to accept only a certain type of argument, you must check the type explicitly using a suitable predicate (see Section 2.6 [Type Predicates], page 27).

### E.6.1 Buffer Internals

Buffers contain fields not directly accessible by the Lisp programmer. We describe them here, naming them by the names used in the C code. Many are accessible indirectly in Lisp programs via Lisp primitives.

Two structures 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 is a list of the `struct buffer_text` fields:

- |            |   |
|------------|---|
| <b>beg</b> | This field contains the actual address of the buffer contents.  |
| <b>gpt</b> | This holds the character position of the gap in the buffer. See Section 27.12 [Buffer Gap], page 495. |

- z** This field contains the character position of the end of the buffer text.
  - gpt\_byte** Contains the byte position of the gap.
  - z\_byte** Holds the byte position of the end of the buffer text.
  - gap\_size** Contains the size of buffer's gap. See Section 27.12 [Buffer Gap], page 495.
  - modiff** This field counts buffer-modification events for this buffer. It is incremented for each such event, and never otherwise changed.
  - save\_modiff**
    - Contains the previous value of **modiff**, as of the last time a buffer was visited or saved in a file.
  - overlay\_modiff**
    - Counts modifications to overlays analogous to **modiff**.
  - beg\_unchanged**
    - Holds the number of characters at the start of the text that are known to be unchanged since the last redisplay that finished.
  - end\_unchanged**
    - Holds the number of characters at the end of the text that are known to be unchanged since the last redisplay that finished.
  - unchanged\_modified**
    - Contains the value of **modiff** at the time of the last redisplay that finished. If this value matches **modiff**, **beg\_unchanged** and **end\_unchanged** contain no useful information.
  - overlay\_unchanged\_modified**
    - Contains the value of **overlay\_modiff** at the time of the last redisplay that finished. If this value matches **overlay\_modiff**, **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** are the other markers referring to this buffer text.
  - intervals**
    - Contains the interval tree which records the text properties of this buffer.
- The fields of **struct buffer** are:
- next** Points to the next buffer, in the chain of all buffers including killed buffers. This chain is used only for garbage collection, in order to collect killed buffers properly. Note that vectors, and most kinds of objects allocated as vectors, are all on one chain, but buffers are on a separate chain of their own.
  - own\_text** This is a **struct buffer\_text** structure. In an ordinary buffer, it holds the buffer contents. In indirect buffers, this field is not used.
  - text** This points to the **buffer\_text** structure that is used 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.

<b>pt</b>	Contains the character position of point in a buffer.
<b>pt_byte</b>	Contains the byte position of point in a buffer.
<b>begv</b>	This field contains the character position of the beginning of the accessible range of text in the buffer.
<b>begv_byte</b>	This field contains the byte position of the beginning of the accessible range of text in the buffer.
<b>zv</b>	This field contains the character position of the end of the accessible range of text in the buffer.
<b>zv_byte</b>	This field contains the byte position of the end of the accessible range of text in the buffer.
<b>base_buffer</b>	In an indirect buffer, this points to the base buffer. In an ordinary buffer, it is null.
<b>local_var_flags</b>	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.)
<b>modtime</b>	This field contains 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 27.5 [Buffer Modification], page 487.
<b>auto_save_modified</b>	This field contains the time when the buffer was last auto-saved.
<b>auto_save_failure_time</b>	The time at which we detected a failure to auto-save, or -1 if we didn't have a failure.
<b>last_window_start</b>	This field contains the <code>window-start</code> position in the buffer as of the last time the buffer was displayed in a window.
<b>clip_changed</b>	This flag is set when narrowing changes in a buffer.
<b>prevent_redisplay_optimizations_p</b>	this flag indicates that redisplay optimizations should not be used to display this buffer.
<b>undo_list</b>	This field points to the buffer's undo list. See Section 32.9 [Undo], page 596.
<b>name</b>	The buffer name is a string that names the buffer. It is guaranteed to be unique. See Section 27.3 [Buffer Names], page 484.

**filename** The name of the file visited in this buffer, or `nil`.

**directory**

The directory for expanding relative file names.

**save\_length**

Length of the file this buffer is visiting, when last read or saved. This and other fields concerned with saving are not kept in the `buffer_text` structure because indirect buffers are never saved.

**auto\_save\_file\_name**

File name used for auto-saving this buffer. This is not in the `buffer_text` because it's not used in indirect buffers at all.

**read\_only**

Non-`nil` means this buffer is read-only.

**mark**

This field contains the mark for the buffer. The mark is a marker, hence it is also included on the list `markers`. See Section 31.7 [The Mark], page 577.

**local\_var alist**

This field contains 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 11.10 [Buffer-Local Variables], page 147.

**major\_mode**

Symbol naming the major mode of this buffer, e.g., `lisp-mode`.

**mode\_name**

Pretty name of major mode, e.g., "Lisp".

**mode\_line\_format**

Mode line element that controls the format of the mode line. If this is `nil`, no mode line will be displayed.

**header\_line\_format**

This field is analogous to `mode_line_format` for the mode line displayed at the top of windows.

**keymap** This field holds the buffer's local keymap. See Chapter 22 [Keymaps], page 347.

**abbrev\_table**

This buffer's local abbrevs.

**syntax\_table**

This field contains the syntax table for the buffer. See Chapter 35 [Syntax Tables], page 684.

**category\_table**

This field contains the category table for the buffer.

**case\_fold\_search**

The value of `case-fold-search` in this buffer.

**tab\_width**

The value of `tab-width` in this buffer.

**fill\_column**

The value of `fill-column` in this buffer.

**left\_margin**

The value of `left-margin` in this buffer.

**auto\_fill\_function**

The value of `auto-fill-function` in this buffer.

**downcase\_table**

This field contains the conversion table for converting text to lower case. See Section 4.9 [Case Tables], page 60.

**upcase\_table**

This field contains the conversion table for converting text to upper case. See Section 4.9 [Case Tables], page 60.

**case\_canon\_table**

This field contains the conversion table for canonicalizing text for case-folding search. See Section 4.9 [Case Tables], page 60.

**case\_eqv\_table**

This field contains the equivalence table for case-folding search. See Section 4.9 [Case Tables], page 60.

**truncate\_lines**

The value of `truncate-lines` in this buffer.

**ctl\_arrow**

The value of `ctl-arrow` in this buffer.

**selective\_display**

The value of `selective-display` in this buffer.

**selective\_display\_ellipsis**

The value of `selective-display-ellipsis` in this buffer.

**minor\_modes**

An alist of the minor modes of this buffer.

**overwrite\_mode**

The value of `overwrite-mode` in this buffer.

**abbrev\_mode**

The value of `abbrev-mode` in this buffer.

**display\_table**

This field contains the buffer's display table, or `nil` if it doesn't have one. See Section 38.21 [Display Tables], page 807.

**save\_modified**

This field contains the time when the buffer was last saved, as an integer. See Section 27.5 [Buffer Modification], page 487.

**mark\_active**

This field is non-`nil` if the buffer's mark is active.

**overlays\_before**

This field holds a list of the overlays in this buffer that end at or before the current overlay center position. They are sorted in order of decreasing end position.

**overlays\_after**

This field holds a list of the overlays in this buffer that end after the current overlay center position. They are sorted in order of increasing beginning position.

**overlay\_center**

This field holds the current overlay center position. See Section 38.9 [Overlays], page 754.

**enable\_multibyte\_characters**

This field holds the buffer's local value of `enable-multibyte-characters`—either `t` or `nil`.

**buffer\_file\_coding\_system**

The value of `buffer-file-coding-system` in this buffer.

**file\_format**

The value of `buffer-file-format` in this buffer.

**auto\_save\_file\_format**

The value of `buffer-auto-save-file-format` in this buffer.

**pt\_marker**

In an indirect buffer, or a buffer that is the base of an indirect buffer, this holds a marker that records point for this buffer when the buffer is not current.

**begv\_marker**

In an indirect buffer, or a buffer that is the base of an indirect buffer, this holds a marker that records `begv` for this buffer when the buffer is not current.

**zv\_marker**

In an indirect buffer, or a buffer that is the base of an indirect buffer, this holds a marker that records `zv` for this buffer when the buffer is not current.

**file\_truename**

The truename of the visited file, or `nil`.

**invisibility\_spec**

The value of `buffer-invisibility-spec` in this buffer.

**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.

**display\_count**

This field is incremented each time the buffer is displayed in a window.

**left\_margin\_width**

The value of `left-margin-width` in this buffer.

**right\_margin\_width**

The value of `right-margin-width` in this buffer.

**indicate\_empty\_lines**

Non-`nil` means indicate empty lines (lines with no text) with a small bitmap in the fringe, when using a window system that can do it.

**display\_time**

This holds a time stamp that is updated each time this buffer is displayed in a window.

**scroll\_up\_aggressively**

The value of `scroll-up-aggressively` in this buffer.

**scroll\_down\_aggressively**

The value of `scroll-down-aggressively` in this buffer.

### E.6.2 Window Internals

Windows have the following accessible fields:

**frame** The frame that this window is on.

**mini\_p** Non-`nil` if this window is a minibuffer window.

**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 a window's parent.

Parent windows do not display buffers, and play little role in display except to shape their child windows. Emacs Lisp programs usually have no access to the parent windows; they operate on the windows at the leaves of the tree, which actually display buffers.

The following four fields also describe the window tree structure.

**hchild** In a window subdivided horizontally by child windows, the leftmost child. Otherwise, `nil`.

**vchild** In a window subdivided vertically by child windows, the topmost child. Otherwise, `nil`.

**next** The next sibling of this window. It is `nil` in a window that is the rightmost or bottommost of a group of siblings.

**prev** The previous sibling of this window. It is `nil` in a window that is the leftmost or topmost of a group of siblings.

**left** This is the left-hand edge of the window, measured in columns. (The leftmost column on the screen is column 0.)

**top** This is the top edge of the window, measured in lines. (The top line on the screen is line 0.)

**height** The height of the window, measured in lines.

**width** The width of the window, measured in columns. This width includes the scroll bar and fringes, and/or the separator line on the right of the window (if any).

<b>buffer</b>	The buffer that the window is displaying. This may change often during the life of the window.
<b>start</b>	The position in the buffer that is the first character to be displayed in the window.
<b>pointm</b>	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.
<b>force_start</b>	If this flag is non-nil, it says that the window has been scrolled explicitly by the Lisp program. 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.
<b>frozen_window_start_p</b>	This field is set temporarily to 1 to indicate to redisplay that <b>start</b> of this window should not be changed, even if point gets invisible.
<b>start_at_line_beg</b>	Non-nil means current value of <b>start</b> was the beginning of a line when it was chosen.
<b>too_small_ok</b>	Non-nil means don't delete this window for becoming "too small."
<b>height_fixed_p</b>	This field is temporarily set to 1 to fix the height of the selected window when the echo area is resized.
<b>use_time</b>	This is the last time that the window was selected. The function <code>get-lru-window</code> uses this field.
<b>sequence_number</b>	A unique number assigned to this window when it was created.
<b>last_modified</b>	The <code>modiff</code> field of the window's buffer, as of the last time a redisplay completed in this window.
<b>last_overlay_modified</b>	The <code>overlay_modiff</code> field of the window's buffer, as of the last time a redisplay completed in this window.
<b>last_point</b>	The buffer's value of point, as of the last time a redisplay completed in this window.
<b>last_had_star</b>	A non-nil value means the window's buffer was "modified" when the window was last updated.
<b>vertical_scroll_bar</b>	This window's vertical scroll bar.

**left\_margin\_width**

The width of the left margin in this window, or `nil` not to specify it (in which case the buffer's value of `left-margin-width` is used).

**right\_margin\_width**

Likewise for the right margin.

**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 not `nil`.

**window\_end\_bytupos**

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-`nil` value if `window_end_pos` is truly valid. This is `nil` if nontrivial redisplay is preempted since in that case the display that `window_end_pos` was computed for did not get onto the screen.

**redisplay\_end\_trigger**

If redisplay in this window goes beyond this buffer position, it runs the `redisplay-end-trigger-hook`.

**cursor**

A structure describing where the cursor is in this window.

**last\_cursor**

The value of `cursor` as of the last redisplay that finished.

**phys\_cursor**

A structure describing where the cursor of this window physically is.

**phys\_cursor\_type**

The type of 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 on.

**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.

**vscroll**

Vertical scroll amount, in pixels. Normally, this is 0.

**dedicated**

`Non-nil` if this window is dedicated to its buffer.

**display\_table**

The window's display table, or `nil` if none is specified for it.

**update\_mode\_line**

Non-`nil` means this window's mode line needs to be updated.

**base\_line\_number**

The line number of a certain position in the buffer, or `nil`. 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 `nil` meaning none is known.

**regionShowing**

If the region (or part of it) is highlighted in this window, this field holds the mark position that made one end of that region. Otherwise, this field is `nil`.

**column\_number\_displayed**

The column number currently displayed in this window's mode line, or `nil` if column numbers are not being displayed.

**current\_matrix**

A glyph matrix describing the current display of this window.

**desired\_matrix**

A glyph matrix describing the desired display of this window.

### E.6.3 Process Internals

The fields of a process are:

**name** A string, the name of the process.

**command** A list containing the command arguments that were used to start this process.

**filter** A function used to accept output from the process instead of a buffer, or `nil`.

**sentinel** A function called whenever the process receives a signal, or `nil`.

**buffer** The associated buffer of the process.

**pid** An integer, the operating system's process ID.

**childp** A flag, non-`nil` if this is really a child process. It is `nil` for a network connection.

**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-`nil`, killing Emacs while this process is still running does not ask for confirmation about killing the process.

**raw\_status\_low****raw\_status\_high**

These two fields record 16 bits each of the process status returned by the `wait` system call.

**status** The process status, as `process-status` should return it.

**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-`nil` if communication with the subprocess uses a PTY; `nil` if it uses a pipe.

**infd** The file descriptor for input from the process.

**outfd** The file descriptor for output to the process.

**subtty** The file descriptor for the terminal that the subprocess is using. (On some systems, there is no need to record this, so the value is `nil`.)

**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.

**encoding\_carryover**  
Size of carryover in encoding.

**inherit\_coding\_system\_flag**  
Flag to set `coding-system` of the process buffer from the coding system used to decode process output.

## Appendix F Standard Errors

Here is the complete list of the error symbols in standard Emacs, grouped by concept. The list includes each symbol's message (on the `error-message` property of the symbol) and a cross reference to a description of how the error can occur.

Each error symbol has an `error-conditions` property 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 symbol `quit` does not have the condition `error`, because quitting is not considered an error.

See Section 10.5.3 [Errors], page 127, for an explanation of how errors are generated and handled.

`symbol`    *string; reference.*

`error`    `"error"`

See Section 10.5.3 [Errors], page 127.

`quit`    `"Quit"`

See Section 21.10 [Quitting], page 338.

`args-out-of-range`

`"Args out of range"`

This happens when trying to access an element beyond the range of a sequence or buffer.

See Chapter 6 [Sequences Arrays Vectors], page 87, See Chapter 32 [Text], page 581.

`arith-error`

`"Arithmet ic error"`

See Section 3.6 [Arithmetic Operations], page 38.

`beginning-of-buffer`

`"Beginning of buffer"`

See Section 30.2.1 [Character Motion], page 560.

`buffer-read-only`

`"Buffer is read-only"`

See Section 27.7 [Read Only Buffers], page 489.

`coding-system-error`

`"Invalid coding system"`

See Section 33.10.3 [Lisp and Coding Systems], page 650.

`cyclic-function-indirection`

`"Symbol's chain of function indirections\`

`contains a loop"`

See Section 9.1.4 [Function Indirection], page 112.

- cyclic-variable-indirection**
  - "Symbol's chain of variable indirections\\ contains a loop"

See Section 11.14 [Variable Aliases], page 157.
- end-of-buffer**
  - "End of buffer"

See Section 30.2.1 [Character Motion], page 560.
- end-of-file**
  - "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 19.3 [Input Functions], page 271.
- file-already-exists**
  - This is a subcategory of **file-error**.

See Section 25.4 [Writing to Files], page 441.
- 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 25.7 [Changing Files], page 450.
- 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 25 [Files], page 434.
- file-locked**
  - This is a subcategory of **file-error**.  
See Section 25.5 [File Locks], page 442.
- file-supersession**
  - This is a subcategory of **file-error**.  
See Section 27.6 [Modification Time], page 488.
- ftp-error**
  - This is a subcategory of **file-error**, which results from problems in accessing a remote file using `ftp`.  
See section "Remote Files" in *The GNU Emacs Manual*.
- invalid-function**
  - "Invalid function"

See Section 9.1.4 [Function Indirection], page 112.
- invalid-read-syntax**
  - "Invalid read syntax"

See Section 2.1 [Printed Representation], page 8.

**invalid-regexp**  
"Invalid regexp"  
See Section 34.3 [Regular Expressions], page 663.

**mark-inactive**  
"The mark is not active now"  
See Section 31.7 [The Mark], page 577.

**no-catch** "No catch for tag"  
See Section 10.5.1 [Catch and Throw], page 125.

**scan-error**  
"Scan error"  
This happens when certain syntax-parsing functions find invalid syntax or mismatched parentheses.  
See Section 30.2.6 [List Motion], page 566, and Section 35.6 [Parsing Expressions], page 691.

**search-failed**  
"Search failed"  
See Chapter 34 [Searching and Matching], page 661.

**setting-constant**  
"Attempt to set a constant symbol"  
The values of the symbols `nil` and `t`, and any symbols that start with `:`, may not be changed.  
See Section 11.2 [Variables that Never Change], page 135.

**text-read-only**  
"Text is read-only"  
This is a subcategory of `buffer-read-only`.  
See Section 32.19.4 [Special Properties], page 620.

**undefined-color**  
"Undefined color"  
See Section 29.20 [Color Names], page 552.

**void-function**  
"Symbol's function definition is void"  
See Section 12.8 [Function Cells], page 171.

**void-variable**  
"Symbol's value as variable is void"  
See Section 11.7 [Accessing Variables], page 143.

**wrong-number-of-arguments**  
"Wrong number of arguments"  
See Section 9.1.3 [Classifying Lists], page 112.

**wrong-type-argument**  
"Wrong type argument"  
See Section 2.6 [Type Predicates], page 27.

These kinds of error, which are classified as special cases of `arith-error`, can occur on certain systems for invalid use of mathematical functions.

**domain-error**

"Arithmetic domain error"

See Section 3.9 [Math Functions], page 44.

**overflow-error**

"Arithmetic overflow error"

This is a subcategory of `domain-error`.

See Section 3.9 [Math Functions], page 44.

**range-error**

"Arithmetic range error"

See Section 3.9 [Math Functions], page 44.

**singularity-error**

"Arithmetic singularity error"

This is a subcategory of `domain-error`.

See Section 3.9 [Math Functions], page 44.

**underflow-error**

"Arithmetic underflow error"

This is a subcategory of `domain-error`.

See Section 3.9 [Math Functions], page 44.

## Appendix G Buffer-Local Variables

The table below lists the general-purpose Emacs variables that automatically become buffer-local in each buffer. Most become buffer-local only when set; a few of them are always local in every buffer. Many Lisp packages define such variables for their internal use, but we don't try to list them all here.

Every buffer-specific minor mode defines a buffer-local variable named ‘*modename-mode*’. See Section 23.3.1 [Minor Mode Conventions], page 397. Minor mode variables will not be listed here.

**auto-fill-function**

See Section 32.14 [Auto Filling], page 604.

**buffer-auto-save-file-format**

See Section 25.12 [Format Conversion], page 468.

**buffer-auto-save-file-name**

See Section 26.2 [Auto-Saving], page 476.

**buffer-backed-up**

See Section 26.1.1 [Making Backups], page 471.

**buffer-display-count**

See Section 28.6 [Buffers and Windows], page 505.

**buffer-display-table**

See Section 38.21.2 [Active Display Table], page 809.

**buffer-display-time**

See Section 28.6 [Buffers and Windows], page 505.

**buffer-file-coding-system**

See Section 33.10.2 [Encoding and I/O], page 649.

**buffer-file-format**

See Section 25.12 [Format Conversion], page 468.

**buffer-file-name**

See Section 27.4 [Buffer File Name], page 485.

**buffer-file-number**

See Section 27.4 [Buffer File Name], page 485.

**buffer-file-truename**

See Section 27.4 [Buffer File Name], page 485.

**buffer-file-type**

See Section 33.10.9 [MS-DOS File Types], page 658.

**buffer-invisibility-spec**

See Section 38.6 [Invisible Text], page 748.

**buffer-offer-save**

See Section 27.10 [Killing Buffers], page 493.

**buffer-save-without-query**

See Section 27.10 [Killing Buffers], page 493.

**buffer-read-only**

See Section 27.7 [Read Only Buffers], page 489.

**buffer-saved-size**

See Section 26.2 [Auto-Saving], page 476.

**buffer-undo-list**

See Section 32.9 [Undo], page 596.

**cache-long-line-scans**

See Section 38.3 [Truncation], page 740.

**case-fold-search**

See Section 34.2 [Searching and Case], page 663.

**ctl-arrow**

See Section 38.20 [Usual Display], page 806.

**cursor-type**

See Section 29.3.3.7 [Cursor Parameters], page 536.

**cursor-in-non-selected-windows**

See Section 28.1 [Basic Windows], page 497.

**comment-column**

See section “Comments” in *The GNU Emacs Manual*.

**default-directory**

See Section 25.8.4 [File Name Expansion], page 457.

**defun-prompt-regexp**

See Section 30.2.6 [List Motion], page 566.

**desktop-save-buffer**

See Section 23.7 [Desktop Save Mode], page 424.

**enable-multibyte-characters**

Section 33.1 [Text Representations], page 640.

**fill-column**

See Section 32.12 [Margins], page 602.

**fill-prefix**

See Section 32.12 [Margins], page 602.

**font-lock-defaults**

See Section 23.6.1 [Font Lock Basics], page 413.

**fringe-cursor-alist**

See Section 38.13.3 [Fringe Cursors], page 779.

**fringe-indicator-alist**

See Section 38.13.2 [Fringe Indicators], page 777.

**fringes-outside-margins**

See Section 38.13 [Fringes], page 776.

**goal-column**

See section “Moving Point” in *The GNU Emacs Manual*.

**header-line-format**

See Section 23.4.7 [Header Lines], page 409.

**indicate-buffer-boundaries**

See Section 38.20 [Usual Display], page 806.

**indicate-empty-lines**

See Section 38.20 [Usual Display], page 806.

**left-fringe-width**

See Section 38.13.1 [Fringe Size/Pos], page 776.

**left-margin**

See Section 32.12 [Margins], page 602.

**left-margin-width**

See Section 38.15.4 [Display Margins], page 786.

**line-spacing**

See Section 38.11 [Line Height], page 761.

**local-abbrev-table**

See Section 36.6 [Standard Abbrev Tables], page 704.

**major-mode**

See Section 23.2.4 [Mode Help], page 390.

**mark-active**

See Section 31.7 [The Mark], page 577.

**mark-ring**

See Section 31.7 [The Mark], page 577.

**mode-line-buffer-identification**

See Section 23.4.4 [Mode Line Variables], page 405.

**mode-line-format**

See Section 23.4.2 [Mode Line Data], page 402.

**mode-line-modified**

See Section 23.4.4 [Mode Line Variables], page 405.

**mode-line-process**

See Section 23.4.4 [Mode Line Variables], page 405.

**mode-name**

See Section 23.4.4 [Mode Line Variables], page 405.

**point-before-scroll**

Used for communication between mouse commands and scroll-bar commands.

**right-fringe-width**

See Section 38.13.1 [Fringe Size/Pos], page 776.

**right-margin-width**

See Section 38.15.4 [Display Margins], page 786.

**save-buffer-coding-system**

See Section 33.10.2 [Encoding and I/O], page 649.

**scroll-bar-width**

See Section 38.14 [Scroll Bars], page 781.

**scroll-down-aggressively**

See Section 28.11 [Textual Scrolling], page 515.

**scroll-up-aggressively**

See Section 28.11 [Textual Scrolling], page 515.

**selective-display**

See Section 38.7 [Selective Display], page 750.

**selective-display-ellipses**

See Section 38.7 [Selective Display], page 750.

**tab-width**

See Section 38.20 [Usual Display], page 806.

**truncate-lines**

See Section 38.3 [Truncation], page 740.

**vertical-scroll-bar**

See Section 38.14 [Scroll Bars], page 781.

**window-size-fixed**

See Section 28.15 [Resizing Windows], page 522.

**write-contents-functions**

See Section 25.2 [Saving Buffers], page 437.

## Appendix H Standard Keymaps

The following symbols are used as the names for various keymaps. Some of these exist when Emacs is first started, others are loaded only when their respective mode is used. This is not an exhaustive list.

Several keymaps are used in the minibuffer. See Section 20.6.3 [Completion Commands], page 289.

Almost all of these maps are used as local maps. Indeed, of the modes that presently exist, only Vip mode and Terminal mode ever change the global keymap.

### `apropos-mode-map`

A sparse keymap for `apropos` buffers.

### `Buffer-menu-mode-map`

A full keymap used by Buffer Menu mode.

### `c-mode-map`

A sparse keymap used by C mode.

### `command-history-map`

A full keymap used by Command History mode.

### `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.

### `custom-mode-map`

A full keymap for Custom mode.

### `debugger-mode-map`

A full keymap used by Debugger mode.

### `direc-mode-map`

A full keymap for `direc-mode` buffers.

### `edit-abbrevs-map`

A sparse keymap used in `edit-abbrevs`.

### `edit-tab-stops-map`

A sparse keymap used in `edit-tab-stops`.

### `electric-buffer-menu-mode-map`

A full keymap used by Electric Buffer Menu mode.

### `electric-history-map`

A full keymap used by Electric Command History mode.

### `emacs-lisp-mode-map`

A sparse keymap used by Emacs Lisp mode.

**esc-map** A full keymap for *ESC* (or *Meta*) commands.

**facemenu-menu**

The sparse keymap that displays the Text Properties menu.

**facemenu-background-menu**

The sparse keymap that displays the Background Color submenu of the Text Properties menu.

**facemenu-face-menu**

The sparse keymap that displays the Face submenu of the Text Properties menu.

**facemenu-foreground-menu**

The sparse keymap that displays the Foreground Color submenu of the Text Properties menu.

**facemenu-indentation-menu**

The sparse keymap that displays the Indentation submenu of the Text Properties menu.

**facemenu-justification-menu**

The sparse keymap that displays the Justification submenu of the Text Properties menu.

**facemenu-special-menu**

The sparse keymap that displays the Special Props submenu of the Text Properties menu.

**function-key-map**

The keymap for translating keypad and function keys.

If there are none, then it contains an empty sparse keymap. See Section 22.14 [Translation Keymaps], page 365.

**fundamental-mode-map**

The sparse keymap for Fundamental mode.

It is empty and should not be changed.

**global-map**

The full keymap containing default global key bindings.

Modes should not modify the Global map.

**grep-mode-map**

The keymap for `grep-mode` buffers.

**help-map** The sparse keymap for the keys that follow the help character *C-h*.

**help-mode-map**

The sparse keymap for Help mode.

**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.

**Info-edit-map**

A sparse keymap used by the `e` command of Info.

**Info-mode-map**

A sparse keymap containing Info commands.

**isearch-mode-map**

A keymap that defines the characters you can type within incremental search.

**key-translation-map**

A keymap for translating keys. This one overrides ordinary key bindings, unlike **function-key-map**. See Section 22.14 [Translation Keymaps], page 365.

**kmacro-map**

A sparse keymap for keys that follows the **C-x C-k** prefix search.

**lisp-interaction-mode-map**

A sparse keymap used by Lisp Interaction mode.

**lisp-mode-map**

A sparse keymap used by Lisp mode.

**menu-bar-edit-menu**

The keymap which displays the Edit menu in the menu bar.

**menu-bar-files-menu**

The keymap which displays the Files menu in the menu bar.

**menu-bar-help-menu**

The keymap which displays the Help menu in the menu bar.

**menu-bar-mule-menu**

The keymap which displays the Mule menu in the menu bar.

**menu-bar-search-menu**

The keymap which displays the Search menu in the menu bar.

**menu-bar-tools-menu**

The keymap which displays the Tools menu in the menu bar.

**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 for the user in **C-h b (display-bindings)**, where it describes the main use of the **C-c** prefix key.

**occur-mode-map**

A sparse keymap used by Occur mode.

**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.

**text-mode-map**

A sparse keymap used by Text mode.

**tool-bar-map**

The keymap defining the contents of the tool bar.

**view-mode-map**

A full keymap used by View mode.

## Appendix I Standard Hooks

The following is a list of 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 23.1 [Hooks], page 382, for more information about using hooks.

Every major mode defines a mode 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 23.2.7 [Mode Hooks], page 393. Most minor modes have mode hooks too. Mode hooks are omitted in the list below.

The variables whose names end in ‘`-hooks`’ or ‘`-functions`’ are usually *abnormal hooks*; their values are lists of functions, but these functions are called in a special way (they are passed arguments, or their values are used). The variables whose names end in ‘`-function`’ have single functions as their values.

`activate-mark-hook`

See Section 31.7 [The Mark], page 577.

`after-change-functions`

See Section 32.26 [Change Hooks], page 638.

`after-change-major-mode-hook`

See Section 23.2.7 [Mode Hooks], page 393.

`after-init-hook`

See Section 39.1.2 [Init File], page 813.

`after-insert-file-functions`

See Section 32.19.7 [Saving Properties], page 626.

`after-make-frame-functions`

See Section 29.1 [Creating Frames], page 529.

`after-revert-hook`

See Section 26.3 [Reverting], page 479.

`after-save-hook`

See Section 25.2 [Saving Buffers], page 437.

`auto-fill-function`

See Section 32.14 [Auto Filling], page 604.

`auto-save-hook`

See Section 26.2 [Auto-Saving], page 476.

`before-change-functions`

See Section 32.26 [Change Hooks], page 638.

`before-init-hook`

See Section 39.1.2 [Init File], page 813.

**before-make-frame-hook**

See Section 29.1 [Creating Frames], page 529.

**before-revert-hook**

See Section 26.3 [Reverting], page 479.

**before-save-hook**

See Section 25.2 [Saving Buffers], page 437.

**blink-paren-function**

See Section 38.19 [Blinking], page 805.

**buffer-access-fontify-functions**

See Section 32.19.8 [Lazy Properties], page 627.

**calendar-load-hook**

See Info file ‘emacs-xtra’, node ‘Calendar Customizing’.

**change-major-mode-hook**

See Section 11.10.2 [Creating Buffer-Local], page 149.

**command-line-functions**

See Section 39.1.4 [Command-Line Arguments], page 815.

**comment-indent-function**

See section “Options Controlling Comments” in *the GNU Emacs Manual*.

**compilation-finish-functions**

Functions to call when a compilation process finishes.

**custom-define-hook**

Hook called after defining each customize option.

**deactivate-mark-hook**

See Section 31.7 [The Mark], page 577.

**desktop-after-read-hook**

Normal hook run after a successful `desktop-read`. May be used to show a buffer list. See section “Saving Emacs Sessions” in *the GNU Emacs Manual*.

**desktop-no-desktop-file-hook**

Normal hook run when `desktop-read` can’t find a desktop file. May be used to show a dired buffer. See section “Saving Emacs Sessions” in *the GNU Emacs Manual*.

**desktop-save-hook**

Normal hook run before the desktop is saved in a desktop file. This is useful for truncating history lists, for example. See section “Saving Emacs Sessions” in *the GNU Emacs Manual*.

**diary-display-hook**

See Info file ‘emacs-xtra’, node ‘Fancy Diary Display’.

**diary-hook**

List of functions called after the display of the diary. Can be used for appointment notification.

**disabled-command-function**

See Section 21.13 [Disabling Commands], page 344.

**echo-area-clear-hook**

See Section 38.4.4 [Echo Area Customization], page 745.

**emacs-startup-hook**

See Section 39.1.2 [Init File], page 813.

**find-file-hook**

See Section 25.1.1 [Visiting Functions], page 434.

**find-file-not-found-functions**

See Section 25.1.1 [Visiting Functions], page 434.

**first-change-hook**

See Section 32.26 [Change Hooks], page 638.

**font-lock-beginning-of-syntax-function**

See Section 23.6.8 [Syntactic Font Lock], page 420.

**font-lock-fontify-buffer-function**

See Section 23.6.4 [Other Font Lock Variables], page 418.

**font-lock-fontify-region-function**

See Section 23.6.4 [Other Font Lock Variables], page 418.

**font-lock-mark-block-function**

See Section 23.6.4 [Other Font Lock Variables], page 418.

**font-lock-syntactic-face-function**

See Section 23.6.8 [Syntactic Font Lock], page 420.

**font-lock-unfontify-buffer-function**

See Section 23.6.4 [Other Font Lock Variables], page 418.

**font-lock-unfontify-region-function**

See Section 23.6.4 [Other Font Lock Variables], page 418.

**initial-calendar-window-hook**

See Info file ‘`emacs-xtra`’, node ‘Calendar Customizing’.

**kbd-macro-termination-hook**

See Section 21.15 [Keyboard Macros], page 345.

**kill-buffer-hook**

See Section 27.10 [Killing Buffers], page 493.

**kill-buffer-query-functions**

See Section 27.10 [Killing Buffers], page 493.

**kill-emacs-hook**

See Section 39.2.1 [Killing Emacs], page 817.

**kill-emacs-query-functions**

See Section 39.2.1 [Killing Emacs], page 817.

**lisp-indent-function**  
**list-diary-entries-hook**  
See Info file ‘`emacs-xtra`’, node ‘Fancy Diary Display’.

**mail-setup-hook**  
See section “Mail Mode Miscellany” in *the GNU Emacs Manual*.

**mark-diary-entries-hook**  
See Info file ‘`emacs-xtra`’, node ‘Fancy Diary Display’.

**menu-bar-update-hook**  
See Section 22.17.5 [Menu Bar], page 377.

**minibuffer-setup-hook**  
See Section 20.14 [Minibuffer Misc], page 302.

**minibuffer-exit-hook**  
See Section 20.14 [Minibuffer Misc], page 302.

**mouse-position-function**  
See Section 29.14 [Mouse Position], page 547.

**nongregorian-diary-listing-hook**  
See Info file ‘`emacs-xtra`’, node ‘Hebrew/Islamic Entries’.

**nongregorian-diary-marking-hook**  
See Info file ‘`emacs-xtra`’, node ‘Hebrew/Islamic Entries’.

**occur-hook**  
**post-command-hook**  
See Section 21.1 [Command Overview], page 304.

**pre-abbrev-expand-hook**  
See Section 36.5 [Abbrev Expansion], page 702.

**pre-command-hook**  
See Section 21.1 [Command Overview], page 304.

**print-diary-entries-hook**  
See Info file ‘`emacs-xtra`’, node ‘Diary Customizing’.

**redisplay-end-trigger-functions**  
See Section 28.19 [Window Hooks], page 527.

**scheme-indent-function**

**suspend-hook**  
See Section 39.2.2 [Suspending Emacs], page 817.

**suspend-resume-hook**  
See Section 39.2.2 [Suspending Emacs], page 817.

**temp-buffer-setup-hook**  
See Section 38.8 [Temporary Displays], page 752.

**temp-buffer-show-function**  
See Section 38.8 [Temporary Displays], page 752.

**temp-buffer-show-hook**

See Section 38.8 [Temporary Displays], page 752.

**term-setup-hook**

See Section 39.1.3 [Terminal-Specific], page 814.

**today-visible-calendar-hook**

See Info file ‘emacs-xtra’, node ‘Calendar Customizing’.

**today-invisible-calendar-hook**

See Info file ‘emacs-xtra’, node ‘Calendar Customizing’.

**window-configuration-change-hook**

See Section 28.19 [Window Hooks], page 527.

**window-scroll-functions**

See Section 28.19 [Window Hooks], page 527.

**window-setup-hook**

See Section 38.23 [Window Systems], page 811.

**window-size-change-functions**

See Section 28.19 [Window Hooks], page 527.

**write-contents-functions**

See Section 25.2 [Saving Buffers], page 437.

**write-file-functions**

See Section 25.2 [Saving Buffers], page 437.

**write-region-annotate-functions**

See Section 32.19.7 [Saving Properties], page 626.

# Index

“”	
‘“’ in printing .....	273
‘“’ in strings.....	18
,	
‘,’ (with backquote) .....	179
‘,@ (with backquote) .....	179
#	
‘#\$’ .....	218
‘#?’ syntax .....	171
‘#:’ read syntax .....	14
‘#@count’ .....	218
‘#n#’ read syntax.....	26
‘#n=’ read syntax.....	26
\$	
‘\$’ in display .....	740
‘\$’ in regexp.....	666
%	
%.....	39
‘%’ in format.....	56
&	
‘&’ in replacement .....	677
&optional.....	163
&rest .....	163
,	
‘,’ for quoting.....	116
(	
‘(’ in regexp.....	669
‘(...’ in lists.....	15
)	
‘)’ in regexp.....	669
*	
*.....	39
‘*’ in <code>interactive</code> .....	306
‘*’ in regexp.....	664
‘*scratch*’ .....	389
+	
+.....	38
‘+’ in regexp.....	664
,	
‘,’ (with backquote) .....	179
‘,@ (with backquote) .....	179
-	
-.....	38
.	
‘.’ in lists .....	16
‘.’ in regexp.....	664
‘.emacs’.....	813
/	
/.....	39
/=.....	35
;	
‘;’ in comment .....	9
<	
<.....	35
<=.....	36
=	
=.....	35
>	
>.....	36
>=.....	36
?	
‘?’ in character constant .....	10
? in minibuffer .....	281
‘?’ in regexp.....	665
@	
‘@’ in <code>interactive</code> .....	306
[	
‘[’ in regexp.....	665
[...] (Edebug) .....	261

]

']' in regexp..... 665

^

'^' in regexp..... 666

‘

'..... 179  
' (list substitution)..... 179

\

'\' in character constant..... 11  
'\' in display..... 740  
'\' in printing..... 273  
'\' in regexp..... 666  
'\' in replacement..... 677  
'\' in strings..... 18  
'\' in symbols..... 13  
'\'' in regexp..... 670  
'\<' in regexp..... 671  
'\=' in regexp..... 670  
'\>' in regexp..... 671  
'\\_<' in regexp..... 671  
'\\_>' in regexp..... 671  
'\'' in regexp..... 670  
'\a'..... 10  
'\b'..... 10  
'\b' in regexp..... 670  
'\B' in regexp..... 671  
'\e'..... 10  
'\f'..... 10  
'\n'..... 10  
'\n' in print..... 276  
'\n' in replacement..... 677  
'\r'..... 10  
'\s'..... 10  
'\s' in regexp..... 670  
'\S' in regexp..... 670  
'\t'..... 10  
'\v'..... 10  
'\w' in regexp..... 670  
'\W' in regexp..... 670

|

'|' in regexp..... 668

1

1+..... 38  
1-..... 38  
1value..... 266**2**

2C-mode-map..... 352

**A**abbrev..... 699  
abbrev tables in modes..... 386  
abbrev-all-caps..... 702  
abbrev-expansion..... 702  
abbrev-file-name..... 701  
abbrev-mode..... 699  
abbrev-prefix-mark..... 702  
abbrev-start-location..... 703  
abbrev-start-location-buffer..... 703  
abbrev-symbol..... 702  
abbrev-table-name-list..... 700  
abbreviate-file-name..... 457  
abrevs-changed..... 702  
abnormal hook..... 382  
abort-recursive-edit..... 343  
aborting..... 342  
abs..... 36  
absolute file name..... 455  
accept input from processes..... 721  
accept-change-group..... 637  
accept-process-output..... 721  
access-file..... 445  
accessibility of a file..... 443  
accessible portion (of a buffer)..... 569  
accessible-keymaps..... 368  
acos..... 44  
action (button property)..... 797  
action, customization keyword..... 198  
activate-change-group..... 637  
activate-mark-hook..... 579  
activating advice..... 230  
active display table..... 809  
active keymap..... 353  
active-minibuffer-window..... 300  
ad-activate..... 230  
ad-activate-all..... 231  
ad-activate-regexp..... 231  
ad-add-advice..... 230  
ad-deactivate..... 231  
ad-deactivate-all..... 231  
ad-deactivate-regexp..... 231  
ad-default-compilation-action..... 231  
ad-define-subr-args..... 235  
ad-disable-advice..... 232  
ad-disable-regexp..... 232  
ad-do-it..... 229  
ad-enable-advice..... 232  
ad-enable-regexp..... 232  
ad-get-arg..... 234  
ad-get-args..... 234  
ad-return-value..... 227  
ad-set-arg..... 234  
ad-set-args..... 234

ad-start-advice.....	231	arith-error example .....	131
ad-stop-advice.....	231	arith-error in division .....	39
ad-unadvise .....	229	arithmetic operations .....	38
ad-unadvise-all.....	229	arithmetic shift .....	42
ad-update.....	231	around-advice .....	227
ad-update-all.....	231	array .....	89
ad-update-regexp.....	231	array elements .....	90
adaptive-fill-first-line-regexp .....	604	arrayp .....	90
adaptive-fill-function.....	604	ASCII character codes .....	10
adaptive-fill-mode .....	603	ascii-case-table.....	61
adaptive-fill-regexp.....	604	aset .....	90
add-hook.....	383	ash.....	42
add-name-to-file.....	450	asin .....	44
add-text-properties.....	616	ask-user-about-lock .....	443
add-to-history.....	283	ask-user-about-supersession-threat .....	489
add-to-invisibility-spec .....	749	asking the user questions .....	296
add-to-list .....	71	assoc .....	82
add-to-ordered-list .....	72	assoc-default .....	83
address field of register .....	14	assoc-string .....	54
adjust-window-trailing-edge.....	523	association list .....	81
adjusting point .....	315	assq .....	82
advice, activating.....	230	assq-delete-all .....	84
advice, defining.....	227	asynchronous subprocess .....	710
advice, enabling and disabling .....	232	atan .....	45
advice, preactivating.....	232	atom .....	64
advising functions .....	226	atomic changes .....	637
advising primitives .....	234	atoms .....	14
after-advice .....	227	attributes of text .....	615
after-change-functions.....	638	Auto Fill mode .....	604
after-change-major-mode-hook .....	393	auto-coding-functions .....	654
after-find-file.....	437	auto-coding-regexp-alist .....	653
after-init-hook.....	814	auto-fill-chars .....	605
after-insert-file-functions.....	626	auto-fill-function .....	605
after-load-alist.....	213	auto-hscroll-mode .....	519
after-make-frame-functions.....	530	auto-mode-alist .....	390
after-revert-hook .....	480	auto-raise-tool-bar-buttons .....	380
after-save-hook.....	439	auto-resize-tool-bar .....	380
after-string (overlay property) .....	759	auto-save-default .....	478
alist .....	81	auto-save-file-name-p .....	476
alist vs. plist .....	107	auto-save-hook .....	478
all-completions .....	287	auto-save-interval .....	477
alt characters.....	13	auto-save-list-file-name .....	479
and .....	122	auto-save-list-file-prefix .....	479
anonymous function .....	170	auto-save-mode .....	476
apostrophe for quoting .....	116	auto-save-timeout .....	478
append .....	68	auto-save-visited-file-name .....	477
append-to-file.....	441	auto-window-vscroll .....	518
apply .....	168	autoload .....	206
apply, and debugging .....	244	autoload cookie .....	207
apropos .....	431	autoload errors .....	207
apropos-mode-map .....	899	automatic face assignment .....	773
aref .....	90	automatically buffer-local .....	148
args, customization keyword .....	197		
argument binding .....	163		
argument lists, features .....	163		
arguments for shell commands .....	706		
arguments, interactive entry .....	305		
arguments, reading .....	278		
		B	
		back-to-indentation .....	613
		backquote (list substitution) .....	179
		backslash in character constant .....	11

backslash in strings.....	18	beginning-of-line .....	562
backslash in symbols.....	13	bell .....	810
backspace .....	10	bell character.....	10
backtrace.....	244	'benchmark.el'.....	861
backtrace-debug.....	244	benchmarking .....	861
backtrace-frame.....	245	big endian.....	732
backtracking .....	262	binary files and text files.....	658
backup file .....	471	bindat-get-field.....	734
backup files, rename or copy .....	473	bindat-ip-to-string.....	735
backup-buffer .....	471	bindat-length.....	734
backup-by-copying .....	473	bindat-pack .....	735
backup-by-copying-when-linked.....	473	bindat-unpack .....	734
backup-by-copying-when-mismatch .....	473	binding arguments.....	163
backup-by-copying-when-privileged-mismatch .....	473	binding local variables.....	136
backup-directory-alist.....	472	binding of a key.....	348
backup-enable-predicate .....	472	bitmap-spec-p .....	767
backup-file-name-p .....	475	bitmaps, fringe .....	779
backup-inhibited.....	472	bitwise arithmetic .....	41
backups and auto-saving.....	471	blink-cursor-alist .....	537
backward-button.....	800	blink-matching-delay .....	806
backward-char .....	561	blink-matching-open .....	806
backward-delete-char-untabify.....	588	blink-matching-paren .....	805
backward-delete-char-untabify-method.....	589	blink-matching-paren-distance.....	805
backward-list .....	566	blink-paren-function .....	805
backward-prefix-chars.....	691	blinking parentheses .....	805
backward-sexp .....	567	bobp .....	582
backward-to-indentation .....	613	body of function .....	162
backward-word .....	561	bolp .....	582
balanced parenthesis motion .....	566	bool-vector-p .....	95
balancing parentheses .....	805	Bool-vectors .....	95
barf-if-buffer-read-only .....	490	boolean .....	2
base 64 encoding .....	635	booleanp .....	3
base buffer .....	494	boundp .....	139
base coding system .....	648	box diagrams, for lists .....	15
base for reading an integer .....	32	break .....	237
base64-decode-region .....	636	breakpoints (Edebug) .....	250
base64-decode-string .....	636	bucket (in obarray) .....	104
base64-encode-region .....	635	buffer .....	481
base64-encode-string .....	635	buffer contents .....	581
basic code (of input character) .....	315	buffer file name .....	485
batch mode .....	836	buffer input stream .....	268
batch-byte-compile .....	217	buffer internals .....	880
baud-rate .....	834	buffer list .....	490
beep .....	810	buffer modification .....	487
before point, insertion .....	585	buffer names .....	484
before-advice .....	227	buffer output stream .....	271
before-change-functions .....	638	buffer text notation .....	4
before-init-hook .....	814	buffer, read-only .....	489
before-make-frame-hook .....	530	buffer-access-fontified-property .....	627
before-revert-hook .....	480	buffer-access-fontify-functions .....	627
before-save-hook .....	439	buffer-auto-save-file-format .....	470
before-string (overlay property) .....	759	buffer-auto-save-file-name .....	476
beginning of line .....	563	buffer-backed-up .....	471
beginning of line in regexp .....	666	buffer-base-buffer .....	495
beginning-of-buffer .....	562	buffer-chars-modified-tick .....	488
beginning-of-defun .....	567	buffer-disable-undo .....	598
beginning-of-defun-function .....	567	buffer-display-count .....	505
		buffer-display-table .....	809

buffer-display-time .....	506	button-put .....	799
buffer-enable-undo .....	598	button-start .....	799
buffer-end .....	560	button-suffix, customization keyword .....	199
buffer-file-coding-system .....	649	button-type .....	799
buffer-file-format .....	469	button-type-get .....	799
buffer-file-name .....	485	button-type-put .....	799
buffer-file-number .....	486	button-type-subtype-p .....	799
buffer-file-truename .....	485	buttons in buffers .....	796
buffer-file-type .....	658	byte compilation .....	214
buffer-has-markers-at .....	575	byte compiler warnings, how to avoid .....	862
buffer-invisibility-spec .....	748	byte packing and unpacking .....	732
buffer-list .....	491	byte-boolean-vars .....	158, 879
buffer-live-p .....	494	byte-code .....	214
buffer-local variables .....	147	byte-code .....	217
buffer-local variables in modes .....	387	byte-code function .....	220
buffer-local variables, general-purpose .....	895	byte-code interpreter .....	217
buffer-local-value .....	150	byte-code-function-p .....	161
buffer-local-variables .....	150	byte-compile .....	215
Buffer-menu-mode-map .....	899	byte-compile-dynamic .....	218
buffer-modified-p .....	487	byte-compile-dynamic-docstrings .....	218
buffer-modified-tick .....	488	byte-compile-file .....	216
buffer-name .....	484	byte-compiling macros .....	177
buffer-name-history .....	284	byte-compiling require .....	209
buffer-offer-save .....	494	byte-recompile-directory .....	216
buffer-read-only .....	490	byte-to-position .....	641
buffer-save-without-query .....	494	bytes .....	47
buffer-saved-size .....	478	bytes and characters .....	645
buffer-size .....	560		
buffer-string .....	584		
buffer-substring .....	582		
buffer-substring-filters .....	583	C	
buffer-substring-no-properties .....	583	C-c .....	352
buffer-undo-list .....	596	C-g .....	338
bufferp .....	481	C-h .....	352
buffers without undo information .....	484	C-M-x .....	247
buffers, controlled in windows .....	505	c-mode-map .....	899
buffers, creating .....	492	c-mode-syntax-table .....	695
buffers, killing .....	493	C-x .....	352
bugs .....	1	C-x 4 .....	352
bugs in this manual .....	1	C-x 5 .....	352
building Emacs .....	870	C-x 6 .....	352
building lists .....	67	C-x RET .....	352
built-in function .....	160	C-x v .....	352
bury-buffer .....	492	C-x X = .....	256
butlast .....	67	caar .....	67
button (button property) .....	797	cache-long-line-scans .....	741
button buffer commands .....	800	cadr .....	67
button properties .....	797	call stack .....	244
button types .....	798	call-interactively .....	310
button-activate .....	799	call-process .....	707
button-at .....	799	call-process-region .....	709
button-down event .....	320	call-process-shell-command .....	710
button-end .....	799	called-interactively-p .....	312
button-face, customization keyword .....	199	calling a function .....	167
button-get .....	799	cancel-change-group .....	637
button-has-type-p .....	799	cancel-debug-on-entry .....	240
button-label .....	799	cancel-timer .....	831
button-prefix, customization keyword .....	199	capitalization .....	59
		capitalize .....	59

capitalize-region .....	613	character insertion.....	587
capitalize-word.....	614	character printing.....	430
car.....	64	character quote.....	686
car-safe.....	65	character sets .....	644
case conversion in buffers.....	613	character to string.....	54
case conversion in Lisp.....	58	character translation tables.....	647
case in replacements.....	676	characters .....	47
case-fold-search.....	663	characters for interactive codes .....	307
case-replace.....	663	characters, multi-byte .....	640
case-table-p.....	61	charset-after.....	646
catch .....	126	charset-bytes.....	645
categories of characters.....	696	charset-dimension.....	645
category (overlay property).....	757	charset-list.....	644
category (text property).....	620	charset-plist.....	644
category table.....	696	charsetp.....	644
category-docstring .....	697	check-coding-system.....	650
category-set-mnemonics .....	698	checkdoc-minor-mode.....	862
category-table.....	697	child process .....	705
category-table-p.....	697	circular list.....	63
cdar .....	67	circular structure, read syntax .....	26
cddr .....	67	c1.....	2
cdr .....	65	CL note—allocate more storage.....	873
cdr-safe.....	65	CL note—case of letters .....	13
ceiling.....	37	CL note—default optional arg .....	163
centering point .....	517	CL note—integers vrs eq .....	35
change hooks .....	638	CL note—interning existing symbol .....	106
change hooks for a character .....	623	CL note—lack union, intersection .....	78
change-major-mode-hook.....	152	CL note—no continuable errors .....	129
changing key bindings .....	361	CL note—only throw in Emacs .....	126
changing to another buffer .....	481	CL note—rplaca vs setcar .....	73
changing window size .....	522	CL note—set local .....	145
char-after .....	581	CL note—special forms compared .....	115
char-before .....	581	CL note—special variables .....	145
char-category-set .....	697	CL note—symbol in obarrays .....	105
char-charset .....	644	class of advice .....	227
char-displayable-p .....	776	cleanup forms .....	133
char-equal .....	52	clear-abbrev-table .....	699
char-or-string-p .....	48	clear-image-cache .....	796
char-property-alist .....	616	clear-string .....	52
char-syntax .....	689	clear-this-command-keys .....	314
char-table length .....	87	clear-visited-file-modtime .....	488
char-table-extra-slot .....	94	click event .....	318
char-table-p .....	93	clickable buttons in buffers .....	796
char-table-parent .....	94	clickable text .....	628
char-table-range .....	94	clipboard support (for MS-Windows) .....	551
char-table-subtype .....	93	clone-indirect-buffer .....	495
char-tables .....	93	close parenthesis character .....	685
char-to-string .....	54	closures not available .....	146
char-valid-p .....	643	clrhash .....	99
char-width .....	760	codes, interactive, description of .....	307
character alternative (in regexp) .....	665	coding conventions in Emacs Lisp .....	856
character arrays .....	47	coding standards .....	856
character as bytes .....	645	coding system .....	648
character case .....	58	coding-system-change-eol-conversion .....	651
character categories .....	696	coding-system-change-text-conversion .....	651
character classes in regexp .....	667	coding-system-eol-type .....	651
character code conversion .....	648	coding-system-for-read .....	655
character codes .....	643	coding-system-for-write .....	656

<code>coding-system-get</code> .....	649	<code>completion-regexp-list</code> .....	287
<code>coding-system-list</code> .....	650	complex arguments .....	278
<code>coding-system-p</code> .....	650	complex command .....	344
color names .....	552	Composite Types (customization) .....	194
<code>color-defined-p</code> .....	553	<code>composition</code> (text property) .....	624
<code>color-gray-p</code> .....	553	<code>composition</code> property, and point display .....	315
<code>color-supported-p</code> .....	553	<code>compute-motion</code> .....	565
<code>color-values</code> .....	553	<code>concat</code> .....	49
colors on text-only terminals .....	554	concatenating lists .....	76
<code>columns</code> .....	609	concatenating strings .....	49
<code>combine-after-change-calls</code> .....	638	<code>cond</code> .....	121
<code>command</code> .....	160	condition name .....	132
command descriptions .....	4	<code>condition-case</code> .....	130
command history .....	344	conditional evaluation .....	120
command in keymap .....	358	conditional selection of windows .....	503
command loop .....	304	<code>cons</code> .....	67
command loop, recursive .....	342	cons cells .....	67
<code>command-debug-status</code> .....	245	<code>cons-cells-consed</code> .....	875
<code>command-error-function</code> .....	129	consing .....	67
<code>command-execute</code> .....	311	<code>consp</code> .....	64
<code>command-history</code> .....	344	constant variables .....	135
<code>command-history-map</code> .....	899	<code>constrain-to-field</code> .....	631
<code>command-line</code> .....	816	continuation lines .....	740
command-line arguments .....	815	<code>continue-process</code> .....	717
command-line options .....	816	control character key constants .....	362
<code>command-line-args</code> .....	816	control character printing .....	430
<code>command-line-functions</code> .....	816	control characters .....	12
<code>command-line-processed</code> .....	816	control characters in display .....	807
<code>command-remapping</code> .....	365	control characters, reading .....	335
<code>command-switch-alist</code> .....	816	control structures .....	119
<code>commandp</code> .....	310	<code>Control-X-prefix</code> .....	352
<code>commandp</code> example .....	292	controller part, model/view/controller .....	804
commands, defining .....	305	conventions for writing major modes .....	385
<code>comment ender</code> .....	687	conventions for writing minor modes .....	397
<code>comment starter</code> .....	687	conversion of strings .....	54
comment syntax .....	687	<code>convert-standard-filename</code> .....	462
comments .....	9	converting numbers .....	36
comments, Lisp convention for .....	865	<code>coordinates-in-window-p</code> .....	525
Common Lisp .....	2	<code>copy-abbrev-table</code> .....	700
<code>compare-buffer-substrings</code> .....	584	<code>copy-alist</code> .....	83
<code>compare-strings</code> .....	53	<code>copy-category-table</code> .....	697
<code>compare-window-configurations</code> .....	527	<code>copy-face</code> .....	772
comparing buffer text .....	584	<code>copy-file</code> .....	451
comparing file modification time .....	488	<code>copy-hash-table</code> .....	101
comparing numbers .....	35	<code>copy-keymap</code> .....	351
compilation (Emacs Lisp) .....	214	<code>copy-marker</code> .....	574
compilation functions .....	215	<code>copy-region-as-kill</code> .....	592
<code>compile-defun</code> .....	216	<code>copy-sequence</code> .....	88
compile-time constant .....	219	<code>copy-syntax-table</code> .....	689
compiled function .....	220	<code>copy-tree</code> .....	70
compiler errors .....	220	copying alists .....	83
complete key .....	348	copying files .....	450
<code>completing-read</code> .....	288	copying lists .....	68
completion .....	285	copying sequences .....	88
completion, file name .....	461	copying strings .....	49
<code>completion-auto-help</code> .....	290	copying vectors .....	92
<code>completion-ignore-case</code> .....	287	<code>cos</code> .....	44
<code>completion-ignored-extensions</code> .....	461	<code>count-lines</code> .....	563

count-loop.....	5	customization types .....	191
count-screen-lines .....	564	customization variables, how to define .....	188
counting columns.....	609	customize-package-emacs-version-alist ...	187
coverage testing .....	266	cut buffer .....	551
coverage testing (Edebug) .....	256	cyclic ordering of windows .....	503
create-file-buffer .....	436		
create-fontset-from-fontset-spec .....	775		
create-glyph .....	810		
create-image .....	793		
creating and deleting directories .....	464		
creating buffers.....	492		
creating hash tables .....	97		
creating keymaps.....	350		
ctl-arrow.....	807		
ctl-x-4-map .....	352		
ctl-x-5-map .....	352		
ctl-x-map.....	352		
current binding.....	136		
current buffer .....	481		
current buffer mark.....	577		
current buffer point and mark (Edebug) .....	257		
current buffer position.....	559		
current command.....	313		
current stack frame.....	241		
current-active-maps .....	354		
current-buffer.....	483		
current-case-table .....	61		
current-column.....	609		
current-fill-column.....	602		
current-frame-configuration.....	546		
current-global-map .....	356		
current-idle-time .....	831		
current-indentation.....	610		
current-input-method.....	659		
current-input-mode .....	833		
current-justification.....	601		
current-kill.....	594		
current-left-margin.....	602		
current-local-map .....	356		
current-message.....	742		
current-minor-mode-maps .....	356		
current-prefix-arg .....	341		
current-time.....	824		
current-time-string.....	824		
current-time-zone .....	824		
current-window-configuration.....	526		
current-word.....	584		
cursor.....	511		
cursor (text property) .....	623		
cursor, fringe.....	779		
cursor-in-echo-area .....	745		
cursor-in-non-selected-windows .....	497		
cursor-type .....	537		
cust-print .....	254		
custom-add-frequent-value .....	191		
customization definitions .....	185		
customization groups, defining .....	187		
customization keywords.....	185		

default-file-modes .....	452	defvaralias .....	157
default-fill-column .....	602	delay-mode-hooks .....	393
default-frame-alist .....	532	delete .....	80
default-fringe-indicator-alist .....	778	delete-and-extract-region .....	588
default-fringes-cursor-alist .....	779	delete-auto-save-file-if-necessary .....	478
default-header-line-format .....	409	delete-auto-save-files .....	478
default-Indicate-buffer-boundaries .....	778	delete-backward-char .....	588
default-input-method .....	659	delete-blank-lines .....	591
default-justification .....	601	delete-char .....	588
default-line-spacing .....	762	delete-directory .....	464
default-major-mode .....	389	delete-dups .....	81
default-minibuffer-frame .....	543	delete-exited-processes .....	712
default-mode-line-format .....	407	delete-field .....	631
default-process-coding-system .....	654	delete-file .....	452
default-text-properties .....	616	delete-frame .....	541
default-truncate-lines .....	740	delete-frame event .....	322
default-value .....	152	delete-frame-functions .....	541
'default.el' .....	812	delete-horizontal-space .....	589
defconst .....	140	delete-indentation .....	589
defcustom .....	188	delete-minibuffer-contents .....	301
defface .....	763	delete-old-versions .....	474
defgroup .....	187	delete-other-windows .....	501
defimage .....	793	delete-overlay .....	755
define customization group .....	187	delete-process .....	712
define customization options .....	188	delete-region .....	588
define hash comparisons .....	99	delete-to-left-margin .....	603
define-abbrev .....	700	delete-window .....	501
define-abbrev-table .....	700	delete-windows-on .....	501
define-button-type .....	798	deleting files .....	450
define-category .....	697	deleting frames .....	541
define-derived-mode .....	391	deleting list elements .....	78
define-fringe-bitmap .....	780	deleting previous char .....	588
define-generic-mode .....	392	deleting processes .....	712
define-globalized-minor-mode .....	401	deleting text vs killing .....	587
define-hash-table-test .....	100	deleting whitespace .....	589
define-key .....	362	deleting windows .....	501
define-key-after .....	381	delq .....	78
define-logical-name .....	452	derived mode .....	391
define-minor-mode .....	399	describe characters and events .....	429
define-obsolete-function-alias .....	173	describe-bindings .....	370
define-obsolete-variable-alias .....	158	describe-buffer-case-table .....	62
define-prefix-command .....	353	describe-categories .....	698
defined-colors .....	553	describe-current-display-table .....	808
defining a function .....	165	describe-display-table .....	808
defining advice .....	227	describe-mode .....	390
defining commands .....	305	describe-prefix-bindings .....	432
defining Lisp variables in C .....	879	description for interactive codes .....	307
defining menus .....	370	description format .....	4
defining-kbd-macro .....	346	deserializing .....	732
definitions of symbols .....	103	desktop save mode .....	424
defmacro .....	178	desktop-buffer-mode-handlers .....	424
defsubr, Lisp symbol for a primitive .....	878	desktop-save-buffer .....	424
defsubst .....	173	destroy-fringe-bitmap .....	780
defun .....	165	destructive list operations .....	73
DEFUN, C macro to define Lisp primitives .....	877	detect-coding-region .....	651
defun-prompt-regexp .....	567	detect-coding-string .....	652
defvar .....	139	diagrams, boxed, for lists .....	15
DEFVAR_INT, DEFVAR_LISP, DEFVAR_BOOL .....	879	dialog boxes .....	549

digit-argument.....	342	display-visual-class.....	557
dimension (of character set) .....	645	display-warning.....	746
ding.....	810	displaying a buffer.....	506
directory name.....	456	displays, multiple.....	530
directory name abbreviation .....	457	dnd-protocol-alist .....	552
directory part (of file name) .....	453	do-auto-save .....	478
directory-abbrev-alist .....	457	doc, customization keyword.....	199
directory-file-name.....	456	doc-directory .....	428
directory-files.....	463	'DOC-version' (documentation) file .....	426
directory-files-and-attributes .....	463	documentation .....	426
directory-oriented functions.....	462	documentation conventions .....	425
dired-kept-versions .....	474	documentation for major mode .....	390
dired-mode-map .....	899	documentation notation.....	3
disable-command.....	344	documentation of function .....	164
disable-point-adjustment .....	315	documentation strings .....	425
disabled.....	344	documentation strings, conventions and tips ..	862
disabled command.....	344	documentation, keys in .....	428
disabled-command-function .....	344	documentation-property .....	426
disabling advice .....	232	dolist .....	124
disabling undo.....	598	DOS file types .....	658
disassemble.....	221	dotimes .....	125
disassembled byte-code .....	221	dotimes-with-progress-reporter .....	744
discard-input .....	337	dotted list .....	63
discarding input .....	337	dotted lists (Edebug) .....	261
display (overlay property) .....	758	dotted pair notation .....	16
display (text property) .....	783	double-click events .....	320
display feature testing .....	555	double-click-fuzz .....	321
display margins.....	786	double-click-time .....	321
display message in echo area .....	741	double-quote in strings .....	18
display property, and point display .....	315	down-list .....	566
display specification .....	783	downcase .....	58
display table .....	807	downcase-region .....	614
display, abstract .....	800	downcase-word .....	614
display, arbitrary objects .....	800	downcasing in lookup-key .....	331
display-backing-store .....	557	drag event .....	319
display-buffer .....	508	drag-n-drop event .....	323
display-buffer-function .....	511	dribble file .....	833
display-buffer-reuse-frames .....	508	dump-emacs .....	871
display-color-cells .....	558	dumping Emacs .....	870
display-color-p .....	556	dynamic loading of documentation .....	217
display-completion-list .....	290	dynamic loading of functions .....	218
display-graphic-p .....	556	dynamic scoping .....	145
display-grayscale-p .....	556	dynamic-completion-table .....	296
display-images-p .....	556		
display-message-or-buffer .....	742		
display-mm-dimensions-alist .....	557		
display-mm-height .....	557	easy-mm-mode-define-minor-mode .....	400
display-mm-width .....	557	echo area .....	741
display-mouse-p .....	556	echo-area-clear-hook .....	745
display-pixel-height .....	557	echo-strokes .....	745
display-pixel-width .....	557	edbug .....	251
display-planes .....	557	Edebug debugging facility .....	245
display-popup-menus-p .....	556	Edebug execution modes .....	247
display-save-under .....	557	Edebug specification list .....	259
display-screens .....	557	edbug-all-defs .....	264
display-selections-p .....	556	edbug-all-forms .....	264
display-supports-face-attributes-p .....	556	edbug-continue-kbd-macro .....	265
display-table-slot .....	808	edbug-display-freq-count .....	256

## E

easy-mm-mode-define-minor-mode .....	400
echo area .....	741
echo-area-clear-hook .....	745
echo-strokes .....	745
edbug .....	251
Edebug debugging facility .....	245
Edebug execution modes .....	247
Edebug specification list .....	259
edbug-all-defs .....	264
edbug-all-forms .....	264
edbug-continue-kbd-macro .....	265
edbug-display-freq-count .....	256

edebug-eval-macro-args.....	259	end of line in regexp.....	666
edebug-eval-top-level-form.....	247	end-of-buffer.....	562
edebug-global-break-condition.....	265	end-of-defun.....	567
edebug-initial-mode.....	264	end-of-defun-function.....	567
edebug-on-error.....	265	end-of-file.....	271
edebug-on-quit.....	265	end-of-line.....	563
edebug-print-circle.....	255	end-of-line conversion.....	648
edebug-print-length.....	254	endianness .....	732
edebug-print-level.....	255	enlarge-window.....	522
edebug-print-trace-after.....	255	enlarge-window-horizontally.....	523
edebug-print-trace-before.....	255	environment.....	110
edebug-save-displayed-buffer-points.....	264	environment variable access.....	820
edebug-save-windows.....	264	environment variables, subprocesses.....	706
edebug-set-global-break-condition.....	251	eobp .....	582
edebug-setup-hook.....	263	EOL conversion .....	648
edebug-sit-for-seconds.....	248	eolp .....	582
edebug-temp-display-freq-count.....	256	eq.....	30
edebug-test-coverage.....	264	eql.....	35
edebug-trace.....	255, 264	equal .....	30
edebug-tracing.....	255	equality .....	30
edebug-unwrap.....	260	erase-buffer .....	588
edit-abbrevs-map.....	899	error .....	128
edit-and-eval-command.....	282	error cleanup .....	133
edit-tab-stops-map.....	899	error debugging .....	237
editing types .....	23	error description .....	131
editor command loop.....	304	error display .....	741
electric-buffer-menu-mode-map.....	899	error handler .....	129
electric-future-map .....	6	error in debug .....	243
electric-history-map.....	899	error message notation .....	4
element (of list) .....	63	error name .....	132
elements of sequences.....	88	error symbol .....	132
'elp.el'.....	861	error-conditions .....	132
elt.....	88	error-message-string .....	131
Emacs event standard notation .....	429	errors .....	127
emacs-build-time.....	6	ESC .....	361
emacs-lisp-docstring-fill-column .....	425	esc-map .....	352
emacs-lisp-mode-map.....	899	ESC-prefix .....	352
emacs-lisp-mode-syntax-table.....	695	escape (ASCII character) .....	10
emacs-major-version .....	7	escape characters .....	276
emacs-minor-version .....	7	escape characters in printing .....	273
emacs-pid.....	822	escape sequence .....	11
emacs-save-session-functions.....	837	escape-syntax character .....	686
emacs-startup-hook .....	814	eval .....	116
emacs-version .....	6	eval, and debugging .....	244
EMACSLOADPATH environment variable.....	203	eval-after-load .....	212
empty list .....	16	eval-and-compile .....	219
emulation-mode-map-alists .....	358	eval-at-startup .....	871
enable-command .....	344	eval-buffer .....	117
enable-local-eval .....	157	eval-buffer (Edebug) .....	247
enable-local-variables .....	155	eval-current-buffer .....	117
enable-multibyte-characters .....	640	eval-current-buffer (Edebug) .....	247
enable-recursive-minibuffers .....	302	eval-defun (Edebug) .....	247
enabling advice .....	232	eval-expression (Edebug) .....	247
encode-coding-region .....	657	eval-expression-debug-on-error .....	238
encode-coding-string .....	657	eval-expression-print-length .....	277
encode-time .....	826	eval-expression-print-level .....	277
encoding file formats .....	468	eval-minibuffer .....	282
encoding in coding systems .....	656	eval-region .....	117

eval-region (Edebug) .....	247	executing-kbd-macro .....	345
eval-when-compile .....	219	execution speed .....	861
evaluated expression argument.....	309	exit .....	342
evaluation.....	110	exit recursive editing .....	342
evaluation error .....	137	exit-minibuffer.....	300
evaluation list group.....	253	exit-recursive-edit .....	343
evaluation notation.....	3	exiting Emacs .....	817
evaluation of buffer contents.....	117	exp .....	45
evaporate (overlay property).....	759	expand-abbrev .....	702
even-window-heights .....	509	expand-file-name .....	457
event printing.....	430	expansion of file names .....	457
event type.....	324	expansion of macros .....	176
event, reading only one .....	331	expression .....	110
event-basic-type.....	325	expression prefix .....	686
event-click-count .....	321	expt .....	45
event-convert-list .....	326	extended-command-history .....	284
event-end .....	326	extent .....	145
event-modifiers .....	324	extra slots of char-table.....	93
event-start .....	326	extra-keyboard-modifiers .....	333
eventp .....	315		
events .....	315		
ewoc .....	800		
ewoc-buffer .....	802	face (button property) .....	797
ewoc-collect .....	803	face (overlay property) .....	757
ewoc-create .....	801	face (text property) .....	620
ewoc-data .....	802	face alias .....	773
ewoc-delete .....	803	face attributes .....	765
ewoc-enter-after .....	802	face codes of text .....	620
ewoc-enter-before .....	802	face id .....	762
ewoc-enter-first .....	802	face-attribute .....	768
ewoc-enter-last .....	802	face-attribute-relative-p .....	768
ewoc-filter .....	803	face-background .....	769
ewoc-get-hf .....	802	face-bold-p .....	770
ewoc-goto-next .....	803	face-differs-from-default-p .....	773
ewoc-goto-node .....	803	face-documentation .....	426, 773
ewoc-goto-prev .....	803	face-equal .....	773
ewoc-invalidate .....	803	face-font .....	770
ewoc-locate .....	802	face-font-family-alternatives .....	771
ewoc-location .....	802	face-font-registry-alternatives .....	771
ewoc-map .....	803	face-font-rescale-alist .....	772
ewoc-next .....	802	face-font-selection-order .....	771
ewoc-nth .....	802	face-foreground .....	769
ewoc-prev .....	802	face-id .....	772
ewoc-refresh .....	803	face-inverse-video-p .....	770
ewoc-set-data .....	802	face-italic-p .....	770
ewoc-set-hf .....	802	face-list .....	772
examining the interactive form .....	306	face-stipple .....	770
examining windows .....	505	face-underline-p .....	770
examples of using interactive .....	309	facemenu-background-menu .....	900
excursion .....	568	facemenu-face-menu .....	900
exec-directory .....	706	facemenu-foreground-menu .....	900
exec-path .....	706	facemenu-indentation-menu .....	900
exec-suffixes .....	705	facemenu-justification-menu .....	900
executable-find .....	450	facemenu-keymap .....	352
execute program .....	705	facemenu-menu .....	900
execute with prefix argument .....	311	facemenu-special-menu .....	900
execute-extended-command .....	311	facep .....	762
execute-kbd-macro .....	345	faces .....	762

faces for font lock .....	419	file-name-coding-system .....	650
faces, automatic choice .....	773	file-name-completion .....	461
false .....	2	file-name-directory .....	453
fboundp .....	172	file-name-extension .....	454
fceiling .....	41	file-name-history .....	284
feature-unload-hook .....	212	file-name-nondirectory .....	454
featurep .....	210	file-name-sans-extension .....	455
features .....	209	file-name-sans-versions .....	454
features .....	210	file-newer-than-file-p .....	445
fetch-bytocode .....	218	file-newest-backup .....	476
ffloor .....	41	file-nlinks .....	448
field (text property) .....	622	file-ownership-preserved-p .....	445
field width .....	57	file-precious-flag .....	439
field-beginning .....	631	file-readable-p .....	444
field-end .....	631	file-regular-p .....	446
field-string .....	631	file-relative-name .....	455
field-string-no-properties .....	631	file-remote-p .....	467
fields .....	630	file-supersession .....	489
fifo data structure .....	85	file-symlink-p .....	445
file accessibility .....	443	file-truename .....	446
file age .....	445	file-writable-p .....	444
file attributes .....	447	fill-column .....	602
file format conversion .....	468	fill-context-prefix .....	603
file hard link .....	450	fill-individual-paragraphs .....	600
file local variables .....	155	fill-individual-varying-indent .....	600
file locks .....	442	fill-nobreak-predicate .....	603
file mode specification error .....	388	fill-paragraph .....	600
file modes and MS-DOS .....	453	fill-paragraph-function .....	601
file modification time .....	445	fill-prefix .....	602
file name completion subroutines .....	461	fill-region .....	600
file name of buffer .....	485	fill-region-as-paragraph .....	600
file name of directory .....	456	fillarray .....	91
file names .....	453	filling text .....	599
file names in directory .....	462	filling, automatic .....	604
file open error .....	437	filter function .....	718
file symbolic links .....	445	filter multibyte flag, of process .....	720
file types on MS-DOS and Windows .....	658	filter-buffer-substring .....	583
file with multiple names .....	450	find file in path .....	449
file, information about .....	443	find library .....	203
file-accessible-directory-p .....	444	find-backup-file-name .....	475
file-already-exists .....	452	find-buffer-visiting .....	486
file-attributes .....	448	find-charset-region .....	646
file-chase-links .....	446	find-charset-string .....	646
file-coding-system-alist .....	653	find-coding-systems-for-Charsets .....	651
file-directory-p .....	446	find-coding-systems-region .....	651
file-error .....	202	find-coding-systems-string .....	651
file-executable-p .....	444	find-file .....	434
file-exists-p .....	443	find-file-hook .....	436
file-expand-wildcards .....	463	find-file-name-handler .....	467
file-local-copy .....	467	find-file-noselect .....	435
file-locked .....	443	find-file-not-found-functions .....	436
file-locked-p .....	442	find-file-other-window .....	435
file-modes .....	447	find-file-read-only .....	436
file-name-absolute-p .....	455	find-file-wildcards .....	436
file-name-all-completions .....	461	find-image .....	794
file-name-all-versions .....	463	find-operation-coding-system .....	655
file-name-as-directory .....	456	finding files .....	434
file-name-buffer-file-type-alist .....	658	finding windows .....	502

first-change-hook .....	639	foo.....	5
fit-window-to-buffer .....	523	for .....	180
fixup-whitespace .....	590	force-mode-line-update .....	402
flags in format specifications .....	58	force-window-update .....	739
float .....	36	forcing redisplay .....	739
float-output-format .....	277	format .....	56
float-time .....	825	format definition .....	468
floating-point functions .....	44	format of keymaps .....	348
floatp .....	34	format specification .....	56
floats-consed .....	876	format, customization keyword .....	198
floor .....	37	format-alist .....	468
flushing input .....	337	format-find-file .....	469
fmakunbound .....	172	format-insert-file .....	469
fn in function's documentation string .....	208	format-mode-line .....	410
focus event .....	322	format-network-address .....	731
focus-follows-mouse .....	545	format-time-string .....	826
follow links .....	629	format-write-file .....	469
follow-link (button property) .....	797	formatting strings .....	56
following-char .....	581	formfeed .....	10
font lock faces .....	419	forms .....	110
Font Lock mode .....	412	forward advice .....	228
font-list-limit .....	774	forward-button .....	800
font-lock-add-keywords .....	417	forward-char .....	560
font-lock-beginning-of-syntax-function ..	420	forward-comment .....	692
font-lock-builtin-face .....	420	forward-line .....	563
font-lock-comment-delimiter-face .....	419	forward-list .....	566
font-lock-comment-face .....	419	forward-sexp .....	566
font-lock-constant-face .....	420	forward-to-indentation .....	613
font-lock-defaults .....	413	forward-word .....	561
font-lock-doc-face .....	419	frame .....	529
font-lock-extend-after-change-region-		frame configuration .....	546
function .....	423	frame parameters .....	531
font-lock-extra-managed-props .....	418	frame size .....	538
font-lock-face (text property) .....	621	frame title .....	540
font-lock-fontify-buffer-function .....	418	frame visibility .....	545
font-lock-fontify-region-function .....	418	frame-background-mode .....	764
font-lock-function-name-face .....	420	frame-char-height .....	539
font-lock-keyword-face .....	419	frame-char-width .....	539
font-lock-keywords .....	414	frame-current-scroll-bars .....	781
font-lock-keywords-case-fold-search .....	417	frame-first-window .....	542
font-lock-keywords-only .....	420	frame-height .....	539
font-lock-mark-block-function .....	418	frame-list .....	541
font-lock-multiline .....	423	frame-live-p .....	541
font-lock-negation-char-face .....	420	frame-local variables .....	153
font-lock-preprocessor-face .....	420	frame-parameter .....	531
font-lock-remove-keywords .....	417	frame-parameters .....	531
font-lock-string-face .....	419	frame-pixel-height .....	539
font-lock-syntactic-face-function .....	421	frame-pixel-width .....	539
font-lock-syntactic-keywords .....	421	frame-selected-window .....	542
font-lock-syntax-table .....	420	frame-title-format .....	541
font-lock-type-face .....	420	frame-visible-p .....	545
font-lock-unfontify-buffer-function .....	418	frame-width .....	539
font-lock-unfontify-region-function .....	418	framep .....	529
font-lock-variable-name-face .....	420	frames, scanning all .....	541
font-lock-warning-face .....	420	free list .....	872
fontification-functions .....	773	frequency counts .....	256
fontified (text property) .....	621	fringe bitmaps .....	779
fonts in this manual .....	2	fringe cursors .....	779

fringe indicators .....	777	generate-new-buffer-name .....	485
fringe-bitmaps-at-pos .....	780	generic characters .....	646
fringe-cursor-alist .....	779	generic comment delimiter .....	687
fringe-indicator-alist .....	778	generic mode .....	392
fringes .....	776	generic string delimiter .....	687
fringes, and empty line indication .....	777	geometry specification .....	540
fringes-outside-margins .....	776	get .....	108
frround .....	41	get-defcustom keyword .....	189
fset .....	172	get-buffer .....	484
ftp-login .....	134	get-buffer-create .....	492
ftruncate .....	41	get-buffer-process .....	718
full keymap .....	348	get-buffer-window .....	505
funcall .....	167	get-buffer-window-list .....	506
funcall, and debugging .....	244	get-char-property .....	615
function .....	160	get-char-property-and-overlay .....	615
function .....	171	get-file-buffer .....	486
function aliases .....	166	get-file-char .....	270
function call .....	113	get-internal-run-time .....	828
function call debugging .....	239	get-largest-window .....	503
function cell .....	102	get-load-suffixes .....	203
function cell in autoload .....	206	get-lru-window .....	502
function definition .....	165	get-process .....	712
function descriptions .....	4	get-register .....	634
function form evaluation .....	113	get-text-property .....	615
function input stream .....	269	get-unused-category .....	697
function invocation .....	167	get-window-with-predicate .....	503
function keys .....	316	getenv .....	820
function name .....	165	gethash .....	99
function output stream .....	272	GIF .....	792
function quoting .....	171	global binding .....	136
function safety .....	174	global break condition .....	251
function-documentation .....	425	global keymap .....	353
function-key-map .....	365	global variable .....	135
functionals .....	168	global-abbrev-table .....	704
functionp .....	161	global-disable-point-adjustment .....	315
functions in modes .....	385	global-key-binding .....	361
functions, making them interactive .....	305	global-map .....	356
Fundamental mode .....	384	global-mode-string .....	407
fundamental-mode .....	388	global-set-key .....	367
fundamental-mode-abbrev-table .....	704	global-unset-key .....	368
fundamental-mode-map .....	900	glyph .....	809

## G

gamma correction .....	537
gap-position .....	496
gap-size .....	496
garbage collection .....	872
garbage collection protection .....	876
garbage-collect .....	873
garbage-collection-messages .....	874
gc-cons-percentage .....	875
gc-cons-threshold .....	874
gc-elapsed .....	875
GCPRO and UNGCPRO .....	878
gcs-done .....	875
generate characters in charsets .....	645
generate-new-buffer .....	493

## H

hack-local-variables .....	155
handle-switch-frame .....	544
handling errors .....	129
hash code .....	99
hash notation .....	8
hash tables .....	97
hash-table-count .....	101

hash-table-p .....	100	ignored-local-variables .....	156
hash-table-rehash-size .....	101	image cache .....	796
hash-table-rehash-threshold .....	101	image descriptor .....	788
hash-table-size .....	101	image-cache-eviction-delay .....	796
hash-table-test .....	101	image-library-alist .....	788
hash-table-weakness .....	101	image-load-path .....	794
hashing .....	104	image-load-path-for-library .....	794
header comments .....	867	image-mask-p .....	791
header line (of a window) .....	409	image-size .....	796
header-line prefix key .....	331	image-type-available-p .....	788
header-line-format .....	409	image-types .....	788
help for major mode .....	390	images in buffers .....	787
help-char .....	431	Imenu .....	410
help-command .....	431	imenu-add-to-menubar .....	410
help-echo (overlay property) .....	758	imenu-case-fold-search .....	411
help-echo (text property) .....	621	imenu-create-index-function .....	412
help-echo event .....	323	imenu-extract-index-name-function .....	412
help-echo, customization keyword .....	199	imenu-generic-expression .....	410
help-event-list .....	432	imenu-prev-index-position-function .....	412
help-form .....	432	imenu-syntax-alist .....	411
help-index (button property) .....	797	implicit progn .....	119
help-map .....	431	inc .....	176
help-mode-map .....	900	indent-according-to-mode .....	610
Helper-describe-bindings .....	432	indent-code-rigidly .....	612
Helper-help .....	432	indent-for-tab-command .....	610
Helper-help-map .....	900	indent-line-function .....	610
hex numbers .....	32	indent-region .....	611
hidden buffers .....	484	indent-region-function .....	611
history list .....	282	indent-relative .....	612
history of commands .....	344	indent-relative-maybe .....	612
history-add-new-input .....	283	indent-rigidly .....	611
history-delete-duplicates .....	284	indent-tabs-mode .....	610
history-length .....	283	indent-to .....	610
HOME environment variable .....	705	indent-to-left-margin .....	603
hook variables, list of .....	903	indentation .....	609
hooks .....	382	indicate-buffer-boundaries .....	777
hooks for changing a character .....	623	indicate-empty-lines .....	777
hooks for loading .....	212	indicators, fringe .....	777
hooks for motion of point .....	623	indirect buffers .....	494
hooks for text changes .....	638	indirect specifications .....	261
hooks for window operations .....	527	indirect-function .....	113
horizontal position .....	609	indirect-variable .....	158
horizontal scrolling .....	518	indirection for functions .....	112
horizontal-scroll-bar prefix key .....	331	infinite loops .....	239
hyper characters .....	13	infinite recursion .....	137
hyperlinks in documentation strings .....	864	infinity .....	33
<b>I</b>			
icon-title-format .....	541	Info-edit-map .....	900
iconified frame .....	545	Info-mode-map .....	901
iconify-frame .....	545	inherit standard syntax .....	687
iconify-frame event .....	322	inheritance of text properties .....	625
identity .....	168	inheriting a keymap's bindings .....	351
idleness .....	831	inhibit-default-init .....	814
IEEE floating point .....	33	inhibit-eol-conversion .....	656
if .....	120	inhibit-field-text-motion .....	561
ignore .....	168	inhibit-file-name-handlers .....	467
		inhibit-file-name-operation .....	467
		inhibit-modification-hooks .....	639
		inhibit-point-motion-hooks .....	624

inhibit-quit.....	339	integer to string .....	55
inhibit-read-only .....	490	integer-or-marker-p .....	573
inhibit-startup-echo-area-message .....	813	integerp .....	34
inhibit-startup-message .....	813	integers .....	32
init file.....	813	integers in specific radix .....	32
init-file-user.....	822	interactive .....	305
initial-frame-alist .....	532	interactive call .....	310
initial-major-mode .....	389	interactive code description .....	307
initialization of Emacs .....	812	interactive completion .....	307
initialize, defcustom keyword .....	189	interactive function .....	305
inline functions .....	173	interactive, examples of using .....	309
innermost containing parentheses .....	693	interactive-form .....	306
input events .....	315	interactive-p .....	311
input focus .....	543	intern .....	105
input methods .....	659	intern-soft .....	106
input modes .....	832	internals, of buffer .....	880
input stream .....	268	internals, of process .....	889
input-method-alist .....	659	internals, of window .....	886
input-method-function .....	334	interning .....	104
input-pending-p .....	336	interpreter .....	110
insert .....	585	interpreter-mode-alist .....	389
insert-abbrev-table-description .....	700	interprogram-cut-function .....	595
insert-and-inherit .....	626	interprogram-paste-function .....	595
insert-before-markers .....	585	interrupt Lisp functions .....	338
insert-before-markers-and-inherit .....	626	interrupt-process .....	716
insert-behind-hooks (overlay property) .....	758	intervals .....	632
insert-behind-hooks (text property) .....	623	intervals-consed .....	876
insert-buffer .....	587	introduction sequence (of character) .....	645
insert-buffer-substring .....	586	invalid prefix key error .....	362
insert-buffer-substring-as-yank .....	592	invalid-function .....	112
insert-buffer-substring-no-properties .....	586	invalid-read-syntax .....	8
insert-button .....	798	invalid-regexp .....	671
insert-char .....	586	invert-face .....	769
insert-default-directory .....	295	invisible (overlay property) .....	759
insert-directory .....	463	invisible (text property) .....	622
insert-directory-program .....	464	invisible frame .....	545
insert-file-contents .....	440	invisible text .....	748
insert-file-contents-literally .....	441	invisible/intangible text, and point .....	315
insert-for-yank .....	592	invocation-directory .....	821
insert-image .....	795	invocation-name .....	821
insert-in-front-hooks (overlay property) .....	758	isearch-mode-map .....	901
insert-in-front-hooks (text property) .....	623	iteration .....	124
insert-register .....	635		
insert-sliced-image .....	795		
insert-text-button .....	799		
inserting killed text .....	593	J	
insertion before point .....	585	joining lists .....	76
insertion of text .....	585	just-one-space .....	590
insertion type of a marker .....	576	justify-current-line .....	601
inside comment .....	693		
inside string .....	693		
installation-directory .....	821	K	
int-to-string .....	55	kbd .....	347
intangible (overlay property) .....	759	kbd-macro-termination-hook .....	346
intangible (text property) .....	622	kept-new-versions .....	474
integer to decimal .....	55	kept-old-versions .....	474
integer to hexadecimal .....	57	key .....	347
integer to octal .....	56	key binding .....	348
		key binding, conventions for .....	859

key lookup .....	358	kill-ring .....	596
key sequence .....	347	kill-ring-max .....	596
key sequence error .....	362	kill-ring-yank-pointer .....	596
key sequence input .....	330	killing buffers .....	493
key translation function .....	366	killing Emacs .....	817
<b>key-binding</b> .....	354	<b>kmacro-map</b> .....	901
<b>key-description</b> .....	429		
<b>key-translation-map</b> .....	366		
keyboard events .....	315		
keyboard events in strings .....	328		
keyboard input .....	329		
keyboard input decoding on X .....	660		
keyboard macro execution .....	311		
keyboard macro termination .....	810		
keyboard macro, terminating .....	337		
keyboard macros .....	345		
keyboard macros (Edebug) .....	248		
<b>keyboard-coding-system</b> .....	657		
<b>keyboard-quit</b> .....	340		
<b>keyboard-translate</b> .....	334		
<b>keyboard-translate-table</b> .....	333		
keymap .....	347		
<b>keymap</b> (button property) .....	797		
<b>keymap</b> (overlay property) .....	759		
<b>keymap</b> (text property) .....	622		
keymap entry .....	358		
keymap format .....	348		
keymap in keymap .....	358		
keymap inheritance .....	351		
keymap of character .....	622		
keymap of character (and overlays) .....	759		
keymap prompt string .....	349		
<b>keymap-parent</b> .....	351		
<b>keymap-prompt</b> .....	371		
<b>keymapp</b> .....	350		
keymaps for translating events .....	365		
keymaps in modes .....	386		
keys in documentation strings .....	428		
keys, reserved .....	859		
keystroke .....	347		
keystroke command .....	161		
keyword symbol .....	135		
<b>keyworddp</b> .....	136		
kill command repetition .....	313		
kill ring .....	591		
<b>kill-all-local-variables</b> .....	151		
<b>kill-append</b> .....	595		
<b>kill-buffer</b> .....	493		
<b>kill-buffer-hook</b> .....	494		
<b>kill-buffer-query-functions</b> .....	494		
<b>kill-emacs</b> .....	817		
<b>kill-emacs-hook</b> .....	817		
<b>kill-emacs-query-functions</b> .....	817		
<b>kill-local-variable</b> .....	151		
<b>kill-new</b> .....	594		
<b>kill-process</b> .....	716		
<b>kill-read-only-ok</b> .....	592		
<b>kill-region</b> .....	592		

lines.....	562	local-unset-key.....	368
lines in region .....	563	local-variable-if-set-p .....	150
link, customization keyword .....	185	local-variable-p.....	150
linking files .....	450	locale .....	660
Lisp debugger .....	237	locale-coding-system.....	660
Lisp expression motion .....	566	locale-info .....	660
Lisp history .....	2	locate file in path.....	449
Lisp library .....	201	locate-file .....	449
Lisp nesting error .....	117	locate-library .....	205
Lisp object .....	8	lock file .....	442
Lisp printer .....	274	lock-buffer .....	443
Lisp reader .....	268	log.....	45
lisp-interaction-mode-map .....	901	log10 .....	45
lisp-mode-abbrev-table .....	704	logand .....	43
lisp-mode-map .....	901	logb .....	34
'lisp-mode.el'.....	395	logging echo-area messages .....	744
list .....	68	logical arithmetic .....	41
list elements .....	64	logical shift .....	41
list form evaluation .....	112	logior .....	43
list in keymap .....	358	lognot .....	44
list length .....	87	logxor .....	44
list motion .....	566	looking-at .....	675
list structure .....	63	looking-back .....	675
list-buffers-directory .....	487	lookup tables .....	97
list-charset-chars .....	644	lookup-key .....	360
list-processes .....	712	loops, infinite .....	239
listify-key-sequence .....	336	lower case .....	58
listp .....	64	lower-frame .....	546
lists .....	63	lowering a frame .....	546
lists and cons cells .....	63	lsh .....	41
lists as sets .....	78	lwarn .....	746
literal evaluation .....	111		
little endian .....	732		
ln .....	452		
load .....	201	M	
load error with require .....	209	M-o .....	352
load errors .....	202	M-x .....	311
load, customization keyword .....	186	MacLisp .....	2
load-average .....	822	macro .....	160
load-file .....	202	macro argument evaluation .....	181
load-file-rep-suffixes .....	203	macro call .....	176
load-history .....	211	macro call evaluation .....	113
load-in-progress .....	202	macro compilation .....	215
load-library .....	202	macro descriptions .....	4
load-path .....	203	macro expansion .....	177
load-read-function .....	202	macroexpand .....	177
load-suffixes .....	203	macroexpand-all .....	177
loading .....	201	macros .....	176
loading hooks .....	212	macros, at compile time .....	219
'loadup.el' .....	870	magic autoload comment .....	207
local binding .....	136	magic file names .....	464
local keymap .....	353	magic-fallback-mode-alist .....	390
local variables .....	136	magic-mode-alist .....	389
local-abbrev-table .....	704	mail-host-address .....	820
local-key-binding .....	360	major mode .....	384
local-map (overlay property) .....	759	major mode conventions .....	385
local-map (text property) .....	622	major mode hook .....	387
local-set-key .....	368	major mode keymap .....	354
		major mode, automatic selection .....	388

major-mode .....	391	maphash .....	99
make-abbrev-table .....	699	mapping functions .....	168
make-auto-save-file-name .....	477	margins, display .....	786
make-backup-file-name .....	475	mark .....	577
make-backup-file-name-function .....	472	mark excursion .....	569
make-backup-files .....	471	mark ring .....	577
make-bool-vector .....	95	mark, the .....	577
make-button .....	798	mark-active .....	579
make-byte-code .....	221	mark-even-if-inactive .....	579
make-category-set .....	697	mark-marker .....	577
make-category-table .....	697	mark-ring .....	579
make-char .....	645	mark-ring-max .....	579
make-char-table .....	93	marker argument .....	308
make-directory .....	464	marker garbage collection .....	572
make-display-table .....	807	marker input stream .....	268
make-face .....	772	marker output stream .....	272
make-frame .....	529	marker relocation .....	572
make-frame-invisible .....	545	marker-buffer .....	575
make-frame-on-display .....	530	marker-insertion-type .....	576
make-frame-visible .....	545	marker-position .....	575
make-frame-visible event .....	322	markerp .....	573
make-glyph-code .....	809	markers .....	572
make-hash-table .....	97	markers as numbers .....	572
make-help-screen .....	433	match data .....	676
make-indirect-buffer .....	495	match, customization keyword .....	199
make-keymap .....	350	match-alternatives, customization keyword .....	197
make-list .....	68	match-beginning .....	678
make-local-variable .....	149	match-data .....	679
make-marker .....	574	match-end .....	678
make-network-process .....	727	match-string .....	678
make-obsolete .....	173	match-string-no-properties .....	678
make-obsolete-variable .....	157	mathematical functions .....	44
make-overlay .....	754	max .....	36
make-progress-reporter .....	743	max-image-size .....	796
make-ring .....	84	max-lisp-eval-depth .....	117
make-sparse-keymap .....	350	max-mini-window-height .....	303
make-string .....	48	max-specpdl-size .....	137
make-symbol .....	105	md5 .....	636
make-symbolic-link .....	452	MD5 checksum .....	636
make-syntax-table .....	688	member .....	79
make-temp-file .....	459	member-ignore-case .....	81
make-temp-name .....	460	membership in a list .....	78
make-text-button .....	799	memory allocation .....	872
make-translation-table .....	647	memory usage .....	875
make-variable-buffer-local .....	150	memory-full .....	875
make-variable-frame-local .....	153	memory-limit .....	875
make-vector .....	92	memory-use-counts .....	875
makehash .....	99	memq .....	78
making buttons .....	798	memql .....	79
makunbound .....	138	menu bar .....	377
manipulating buttons .....	799	menu definition example .....	376
map-char-table .....	95	menu keymaps .....	370
map-keymap .....	369	menu prompt string .....	370
map-y-or-n-p .....	298	menu separators .....	374
mapatoms .....	106	menu-bar prefix key .....	331
mapc .....	169	menu-bar-edit-menu .....	901
mapcar .....	169	menu-bar-files-menu .....	901
mapconcat .....	169		

menu-bar-final-items.....	378	minibuffer-setup-hook.....	302
menu-bar-help-menu.....	901	minibuffer-window.....	300
menu-bar-mule-menu.....	901	minibuffer-window-active-p.....	301
menu-bar-search-menu.....	901	minibufferp.....	302
menu-bar-tools-menu.....	901	minimum window size.....	524
menu-bar-update-hook.....	378	minor mode.....	397
menu-item.....	372	minor mode conventions.....	397
menu-prompt-more-char.....	376	minor-mode-alist.....	406
merge-face-attribute.....	768	minor-mode-key-binding.....	361
message.....	741	minor-mode-list.....	397
message digest computation .....	636	minor-mode-map-alist.....	356
message-box.....	742	minor-mode-overriding-map-alist.....	357
message-log-max.....	744	misc-objects-consed.....	876
message-or-box.....	742	mod.....	40
message-truncate-lines.....	745	mode.....	382
meta character key constants.....	362	mode help.....	390
meta character printing.....	430	mode hook.....	387
meta characters .....	12	mode line.....	402
meta characters lookup.....	349	mode loading.....	388
meta-prefix-char.....	361	mode variable.....	397
min.....	36	mode-class (property).....	387
minibuffer.....	278	mode-line construct.....	402
minibuffer completion .....	288	mode-line prefix key.....	331
minibuffer history .....	282	mode-line-buffer-identification.....	405
minibuffer input.....	342	mode-line-format.....	404
minibuffer window, and next-window .....	503	mode-line-frame-identification.....	405
minibuffer windows.....	300	mode-line-modes.....	406
minibuffer-allow-text-properties .....	280	mode-line-modified.....	405
minibuffer-auto-raise.....	546	mode-line-mule-info.....	405
minibuffer-complete.....	290	mode-line-position.....	405
minibuffer-complete-and-exit.....	290	mode-line-process.....	406
minibuffer-complete-word .....	289	mode-name.....	406
minibuffer-completion-confirm.....	289	mode-specific-map.....	352
minibuffer-completion-contents .....	301	model/view/controller.....	800
minibuffer-completion-help.....	290	modification flag (of buffer).....	487
minibuffer-completion-predicate .....	289	modification of lists.....	75
minibuffer-completion-table.....	289	modification time of buffer.....	488
minibuffer-contents.....	301	modification time of file .....	448
minibuffer-contents-no-properties .....	301	modification-hooks (overlay property).....	758
minibuffer-depth.....	302	modification-hooks (text property).....	623
minibuffer-exit-hook.....	302	modifier bits (of input character) .....	315
minibuffer-frame-alist.....	532	modify-all-frames-parameters.....	531
minibuffer-help-form.....	302	modify-category-entry.....	698
minibuffer-history .....	284	modify-frame-parameters .....	531
minibuffer-local-completion-map .....	290	modify-syntax-entry.....	689
minibuffer-local-filename-completion-map .....	291	modulus.....	40
minibuffer-local-map.....	280	momentary-string-display .....	753
minibuffer-local-must-match-filename-map .....	291	most-negative-fixnum .....	33
minibuffer-local-must-match-map .....	291	most-positive-fixnum .....	33
minibuffer-local-ns-map .....	281	motion by chars, words, lines, lists .....	560
minibuffer-message .....	303	motion event.....	321
minibuffer-prompt .....	301	mouse click event.....	318
minibuffer-prompt-end.....	301	mouse drag event.....	319
minibuffer-prompt-width .....	301	mouse events, data in .....	326
minibuffer-scroll-window .....	302	mouse events, in special parts of frame.....	331
minibuffer-selected-window .....	302	mouse events, repeated .....	320
		mouse motion events .....	321
		mouse pointer shape.....	550

mouse position .....	547	newline in print .....	275
mouse position list, accessing .....	326	newline in strings .....	18
mouse tracking .....	547	newline-and-indent .....	610
mouse, availability .....	556	next input .....	335
mouse-1 .....	629	next-button .....	800
mouse-2 .....	859	next-char-property-change .....	619
mouse-action (button property) .....	797	next-frame .....	542
mouse-face (button property) .....	797	next-history-element .....	300
mouse-face (overlay property) .....	758	next-matching-history-element .....	300
mouse-face (text property) .....	621	next-overlay-change .....	760
mouse-movement-p .....	326	next-property-change .....	618
mouse-on-link-p .....	630	next-screen-context-lines .....	517
mouse-pixel-position .....	548	next-single-char-property-change .....	619
mouse-position .....	547	next-single-property-change .....	619
mouse-position-function .....	547	next-window .....	503
move to beginning or end of buffer .....	561	nil .....	2
move-marker .....	576	nil as a list .....	16
move-overlay .....	755	nil in keymap .....	358
move-to-column .....	609	nil input stream .....	269
move-to-left-margin .....	603	nil output stream .....	272
move-to-window-line .....	565	nlistp .....	64
movemail .....	706	no-byte-compile .....	214
MS-DOS and file modes .....	453	no-catch .....	126
MS-DOS file types .....	658	no-redraw-on-reenter .....	739
mule-keymap .....	352	no-self-insert property .....	701
multibyte characters .....	640	non-ASCII characters .....	640
multibyte text .....	640	non-ASCII text in keybindings .....	367
multibyte-char-to-unibyte .....	642	nonascii-insert-offset .....	641
multibyte-string-p .....	641	nonascii-translation-table .....	642
multibyte-syntax-as-symbol .....	695	nondirectory part (of file name) .....	453
multiline font lock .....	422	noninteractive .....	836
multiple windows .....	498	nonlocal exits .....	125
multiple X displays .....	530	nonprinting characters, reading .....	335
multiple-frames .....	541	noreturn .....	267
		normal hook .....	382
		normal-auto-fill-function .....	605
		normal-backup-enable-predicate .....	472
		normal-mode .....	388
		not .....	122
		not-modified .....	488
		notation .....	3
		nreverse .....	76
		nth .....	66
		nthcdr .....	66
		null .....	64
		num-input-keys .....	331
		num-nonmacro-input-events .....	333
		number comparison .....	35
		number conversions .....	36
		number-or-marker-p .....	573
		number-sequence .....	70
		number-to-string .....	55
		numberp .....	34
		numbers .....	32
		numeric prefix argument .....	340
		numeric prefix argument usage .....	308
		numerical RGB color specification .....	552

**O**

obarray ..... 104  
obarray ..... 106  
obarray in completion ..... 286  
object ..... 8  
object internals ..... 880  
object to string ..... 275  
occur-mode-map ..... 901  
octal character code ..... 11  
octal character input ..... 335  
octal numbers ..... 32  
one-window-p ..... 500  
only-global-abbrevs ..... 701  
open parenthesis character ..... 685  
open-dribble-file ..... 833  
open-network-stream ..... 725  
open-paren-in-column-0-is-defun-start ..... 567  
open-termscript ..... 834  
operating system environment ..... 819  
operations (property) ..... 467  
option descriptions ..... 6  
optional arguments ..... 163  
options on command line ..... 816  
options, defcustom keyword ..... 189  
or ..... 123  
ordering of windows, cyclic ..... 503  
other-buffer ..... 491  
other-window ..... 504  
other-window-scroll-buffer ..... 516  
output from processes ..... 717  
output stream ..... 271  
output-controlling variables ..... 276  
overall prompt string ..... 349  
overflow ..... 32  
overflow-newline-into-fringe ..... 779  
overlay-arrow-position ..... 781  
overlay-arrow-string ..... 781  
overlay-arrow-variable-list ..... 781  
overlay-buffer ..... 754  
overlay-end ..... 754  
overlay-get ..... 757  
overlay-properties ..... 757  
overlay-put ..... 757  
overlay-recenter ..... 756  
overlay-start ..... 754  
overlayp ..... 754  
overlays ..... 754  
overlays-at ..... 759  
overlays-in ..... 760  
overriding-local-map ..... 357  
overriding-local-map-menu-flag ..... 357  
overriding-terminal-local-map ..... 357  
overwrite-mode ..... 587

**P**

package-version, customization keyword ..... 186  
packing ..... 732

padding ..... 57  
page-delimiter ..... 683  
paired delimiter ..... 686  
paragraph-separate ..... 683  
paragraph-start ..... 683  
parent of char-table ..... 93  
parent process ..... 705  
parenthesis ..... 15  
parenthesis depth ..... 694  
parenthesis matching ..... 805  
parenthesis mismatch, debugging ..... 265  
parenthesis syntax ..... 685  
parse-colon-path ..... 821  
parse-partial-sexp ..... 694  
parse-sexp-ignore-comments ..... 695  
parse-sexp-lookup-properties ..... 690, 695  
parser state ..... 693  
parsing buffer text ..... 684  
passwords, reading ..... 299  
PATH environment variable ..... 705  
path-separator ..... 821  
PBM ..... 792  
peculiar error ..... 132  
peeking at input ..... 335  
percent symbol in mode line ..... 403  
perform-replace ..... 681  
performance analysis ..... 256  
permanent local variable ..... 152  
permission ..... 447  
piece of advice ..... 226  
pipes ..... 711  
play-sound ..... 835  
play-sound-file ..... 835  
play-sound-functions ..... 835  
plist ..... 107  
plist vs. alist ..... 107  
plist-get ..... 108  
plist-member ..... 109  
plist-put ..... 109  
point ..... 559  
point excursion ..... 569  
point in window ..... 511  
point with narrowing ..... 559  
point-entered (text property) ..... 623  
point-left (text property) ..... 623  
point-marker ..... 574  
point-max ..... 560  
point-max-marker ..... 574  
point-min ..... 559  
point-min-marker ..... 574  
pointer (text property) ..... 623  
pointer shape ..... 550  
pointers ..... 14  
pop ..... 65  
pop-mark ..... 578  
pop-to-buffer ..... 507  
pop-up-frame-alist ..... 509  
pop-up-frame-function ..... 509

pop-up-frames .....	509	previous-single-property-change .....	619
pop-up-windows .....	509	previous-window .....	504
pos-visible-in-window-p .....	514	primitive .....	160
position (in buffer) .....	559	primitive function internals .....	876
position argument .....	307	primitive type .....	8
position in window .....	511	primitive-undo .....	598
position of mouse .....	547	prin1 .....	274
position-bytes .....	641	prin1-to-string .....	275
positive infinity .....	33	princ .....	275
posix-looking-at .....	676	print .....	274
posix-search-backward .....	676	print example .....	272
posix-search-forward .....	676	print name cell .....	102
posix-string-match .....	676	print-circle .....	277
posn-actual-col-row .....	327	print-continuous-numbering .....	277
posn-area .....	326	print-escape-multibyte .....	276
posn-at-point .....	328	print-escape-newlines .....	276
posn-at-x-y .....	328	print-escape-nonascii .....	276
posn-col-row .....	327	print-gensym .....	277
posn-image .....	327	print-help-return-message .....	431
posn-object .....	327	print-length .....	277
posn-object-width-height .....	327	print-level .....	277
posn-object-x-y .....	327	print-number-table .....	277
posn-point .....	326	print-quoted .....	276
posn-string .....	327	printed representation .....	8
posn-timestamp .....	327	printed representation for characters .....	10
posn-window .....	326	printing .....	268
posn-x-y .....	327	printing (Edebug) .....	254
post-command-hook .....	304	printing circular structures .....	254
post-gc-hook .....	874	printing limits .....	277
postscript images .....	792	printing notation .....	3
pre-abbrev-expand-hook .....	703	priority (overlay property) .....	757
pre-command-hook .....	304	process .....	705
preactivating advice .....	232	process filter .....	718
preceding-char .....	582	process filter multibyte flag .....	720
precision in format specifications .....	58	process input .....	714
predicates for numbers .....	34	process internals .....	889
prefix argument .....	340	process output .....	717
prefix argument unreadring .....	335	process sentinel .....	721
prefix command .....	353	process signals .....	715
prefix key .....	352	process-adaptive-read-buffering .....	717
prefix, defgroup keyword .....	188	process-buffer .....	718
prefix-arg .....	341	process-coding-system .....	714
prefix-help-command .....	432	process-coding-system-alist .....	654
prefix-numeric-value .....	341	process-command .....	713
preloading additional functions and variables ..	870	process-connection-type .....	711
prepare-change-group .....	637	process-contact .....	725
preventing backtracking .....	260	process-datagram-address .....	726
preventing prefix key .....	359	process-environment .....	821
preventing quitting .....	339	process-exit-status .....	714
previous complete subexpression .....	693	process-file .....	709
previous-button .....	800	process-filter .....	720
previous-char-property-change .....	619	process-filter-multibyte-p .....	721
previous-frame .....	542	process-get .....	714
previous-history-element .....	300	process-id .....	713
previous-matching-history-element .....	300	process-kill-without-query .....	723
previous-overlay-change .....	760	process-list .....	712
previous-property-change .....	619	process-mark .....	718
previous-single-char-property-change .....	619	process-name .....	713

process-plist.....	714	quietly-read-abbrev-file .....	701
process-put .....	714	quit-flag.....	339
process-query-on-exit-flag.....	723	quit-process.....	716
process-running-child-p .....	715	quitting.....	338
process-send-eof.....	715	quitting from infinite loop.....	239
process-send-region.....	715	quote .....	116
process-send-string.....	715	quote character.....	693
process-sentinel.....	723	quoted character input .....	335
process-status.....	713	quoted-insert suppression .....	364
process-tty-name.....	714	quoting characters in printing.....	273
processor run time.....	828	quoting using apostrophe .....	116
processp.....	705		
profiling.....	861		
prog1 .....	120		
prog2 .....	120		
progn .....	119		
program arguments.....	706		
program directories.....	706		
programmed completion .....	296		
programming conventions.....	860		
programming types.....	9		
progress reporting.....	743		
progress-reporter-done.....	744		
progress-reporter-force-update .....	744		
progress-reporter-update .....	743		
prompt for file name.....	293		
prompt string (of menu).....	370		
prompt string of keymap .....	349		
properties of text .....	615		
propertyize .....	617		
property category of text character .....	620		
property list.....	107		
property list cell.....	102		
property lists vs association lists.....	107		
protect C variables from garbage collection.....	878		
protected forms .....	133		
provide .....	210		
providing features .....	209		
PTYs.....	711		
punctuation character .....	685		
pure storage.....	871		
pure-bytes-used.....	872		
purecopy .....	872		
purify-flag .....	872		
push .....	71		
push-button .....	800		
push-mark.....	578		
put .....	108		
put-image .....	795		
put-text-property .....	616		
puthash .....	99		
<b>Q</b>			
query-replace-history.....	284	read-blanks-input .....	281
query-replace-map .....	682	read-non-nil-coding-system.....	653
querying the user.....	296	read-only (text property) .....	622
question mark in character constant.....	10	read-only buffer .....	489
		read-only buffers in interactive.....	306
		read-only character .....	622

read-passwd .....	299	remhash .....	99
read-quoted-char .....	335	remove .....	80
read-quoted-char quitting .....	339	remove-from-invisibility-spec .....	749
read-string .....	280	remove-hook .....	384
read-variable .....	293	remove-images .....	795
reading .....	268	remove-list-of-text-properties .....	617
reading a single event .....	331	remove-overlays .....	755
reading from files .....	440	remove-text-properties .....	617
reading from minibuffer with completion .....	288	remq .....	79
reading interactive arguments .....	307	rename-auto-save-file .....	478
reading numbers in hex, octal, and binary .....	32	rename-buffer .....	484
reading symbols .....	104	rename-file .....	451
<b>real-last-command</b> .....	313	repeat events .....	320
rearrangement of lists .....	75	repeated loading .....	208
rebinding .....	361	replace bindings .....	363
recent-auto-save-p .....	477	replace characters .....	633
recent-keys .....	833	replace matched text .....	676
recenter .....	517	replace-buffer-in-windows .....	508
record command history .....	310	replace-match .....	676
recording input .....	833	replace-regexp-in-string .....	681
recursion .....	124	replacement after search .....	681
<b>recursion-depth</b> .....	343	require .....	210
recursive command loop .....	342	require, customization keyword .....	186
recursive editing level .....	342	require-final-newline .....	440
recursive evaluation .....	110	requiring features .....	209
recursive minibuffers .....	302	reserved keys .....	859
recursive-edit .....	343	resize frame .....	538
redirect-frame-focus .....	544	resize window .....	522
redisplay .....	740	rest arguments .....	163
redisplay-dont-pause .....	740	restore-buffer-modified-p .....	487
redisplay-end-trigger-functions .....	528	restriction (in a buffer) .....	569
redisplay-preemption-period .....	739	resume (cf. no-redraw-on-reenter) .....	739
redo .....	596	return (ASCII character) .....	10
redraw-display .....	739	reverse .....	70
redraw-frame .....	739	reversing a list .....	76
references, following .....	859	revert-buffer .....	479
regexp .....	663	revert-buffer-function .....	480
regexp alternative .....	668	revert-buffer-insert-file-contents-function .....	480
regexp grouping .....	669	revert-without-query .....	480
regexp searching .....	673	rgb value .....	553
<b>regexp-history</b> .....	284	right-fringe-width .....	776
regexp-opt .....	672	right-margin-width .....	787
regexp-opt-depth .....	673	ring data structure .....	84
regexp-quote .....	672	ring-bell-function .....	810
regexp used standardly in editing .....	683	ring-copy .....	85
region (between point and mark) .....	579	ring-elements .....	85
region argument .....	308	ring-empty-p .....	85
<b>region-beginning</b> .....	580	ring-insert .....	85
<b>region-end</b> .....	580	ring-insert-at-beginning .....	85
register-alist .....	634	ring-length .....	85
registers .....	634	ring-p .....	85
regular expression .....	663	ring-ref .....	85
regular expression searching .....	673	ring-remove .....	85
regular expressions, developing .....	663	ring-size .....	85
<b>reindent-then-newline-and-indent</b> .....	611	risky-local-variable-p .....	156
relative file name .....	455	rm .....	452
remainder .....	39	round .....	37
remapping commands .....	364		

rounding in conversions.....	36	scroll-other-window.....	516
rounding without conversion.....	40	scroll-preserve-screen-position.....	517
rplaca.....	73	scroll-right.....	519
rplacd.....	73	scroll-step.....	517
run time stack.....	244	scroll-up.....	515
run-at-time.....	829	scroll-up-aggressively.....	517
run-hook-with-args.....	383	scrolling textually.....	515
run-hook-with-args-until-failure.....	383	search-backward.....	662
run-hook-with-args-until-success.....	383	search-failed.....	661
run-hooks.....	383	search-forward.....	661
run-mode-hooks.....	393	search-spaces-regexp.....	675
run-with-idle-timer.....	831	searching.....	661
<b>S</b>			
safe local variable .....	156	searching active keymaps for keys.....	355
safe-length .....	66	searching and case.....	663
safe-local-eval-forms .....	157	searching and replacing.....	681
safe-local-variable-p .....	156	searching for regexp.....	673
safe-local-variable-values .....	156	seconds-to-time.....	828
safe-magic (property) .....	466	select safe coding system .....	652
safety of functions .....	174	select-frame .....	544
same-window-buffer-names .....	510	select-frame-set-input-focus .....	544
same-window-p .....	511	select-safe-coding-system .....	652
same-window-regexp .....	511	select-safe-coding-system-accept-default-p .....	652
save-abbrevs .....	701	select-window .....	502
save-buffer .....	437	selected window .....	497
save-buffer-coding-system .....	650	selected-frame .....	543
save-current-buffer .....	483	selected-window .....	502
save-excursion .....	569	selecting a buffer .....	481
save-match-data .....	680	selecting a window .....	502
save-restriction .....	570	selection (for window systems) .....	550
save-selected-window .....	502	selection-coding-system .....	551
save-some-buffers .....	438	selective-display .....	750
save-window-excursion .....	526	selective-display-ellipses .....	751
saving buffers .....	437	self-evaluating form .....	111
saving text properties .....	626	self-insert-and-exit .....	300
saving window information .....	526	self-insert-command .....	587
scalable-fonts-allowed .....	772	self-insert-command override .....	364
scan-lists .....	692	self-insert-command, minor modes .....	399
scan-sexps .....	692	self-insertion .....	587
scope .....	145	send-string-to-terminal .....	834
scrap support (for Mac OS) .....	551	sending signals .....	715
screen layout .....	25	sentence-end .....	683
screen of terminal .....	498	sentence-end-double-space .....	601
screen size .....	538	sentence-end-without-period .....	601
screen-height .....	539	sentence-end-without-space .....	601
screen-width .....	539	sentinel (of process) .....	721
scroll bars .....	781	sequence .....	87
scroll-bar-event-ratio .....	328	sequence length .....	87
scroll-bar-mode .....	782	sequencep .....	87
scroll-bar-scale .....	328	serializing .....	732
scroll-bar-width .....	782	session manager .....	836
scroll-conservatively .....	516	set .....	144
scroll-down .....	515	set, defcustom keyword .....	189
scroll-down-aggressively .....	516	set-after, defcustom keyword .....	190
scroll-left .....	519	set-auto-mode .....	389
scroll-margin .....	516	set-buffer .....	483
		set-buffer-auto-saved .....	477
		set-buffer-major-mode .....	389

set-buffer-modified-p.....	487	set-standard-case-table.....	61
set-buffer-multibyte.....	643	set-syntax-table.....	690
set-case-syntax.....	62	set-terminal-coding-system.....	658
set-case-syntax-delims.....	61	set-text-properties.....	617
set-case-syntax-pair.....	61	set-time-zone-rule.....	824
set-case-table.....	61	set-visited-file-modtime.....	489
set-category-table.....	697	set-visited-file-name.....	486
set-char-table-default.....	94	set-window-buffer.....	505
set-char-table-extra-slot.....	94	set-window-configuration.....	526
set-char-table-parent.....	94	set-window-dedicated-p.....	511
set-char-table-range.....	94	set-window-display-table.....	809
set-default.....	153	set-window-fringes.....	777
set-default-file-modes.....	452	set-window-hscroll.....	520
set-display-table-slot.....	808	set-window-margins.....	787
set-face-attribute.....	767	set-window-point.....	512
set-face-background.....	769	set-window-redisplay-end-trigger.....	528
set-face-bold-p.....	769	set-window-scroll-bars.....	782
set-face-font.....	769	set-window-start.....	513
set-face-foreground.....	769	set-window-vscroll.....	518
set-face-inverse-video-p.....	769	setcar.....	73
set-face-italic-p.....	769	setcdr.....	74
set-face-stipple.....	769	setenv.....	820
set-face-underline-p.....	769	setplist.....	108
set-file-modes.....	452	setprv.....	822
set-file-times.....	452	setq.....	144
set-fontset-font.....	776	setq-default.....	152
set-frame-configuration.....	546	sets.....	78
set-frame-height.....	539	setting modes of files.....	450
set-frame-position.....	539	setting-constant.....	135
set-frame-selected-window.....	543	severity level.....	746
set-frame-size.....	539	sexp motion.....	566
set-frame-width.....	540	shadowing of variables.....	136
set-fringe-bitmap-face.....	780	shallow binding.....	147
set-input-method.....	659	shared structure, read syntax.....	26
set-input-mode.....	832	shell command arguments.....	706
set-keyboard-coding-system.....	657	shell-command-history.....	284
set-keymap-parent.....	351	shell-command-to-string.....	710
set-left-margin.....	602	shell-quote-argument.....	706
set-mark.....	578	show-help-function.....	624
set-marker.....	576	shrink-window.....	523
set-marker-insertion-type.....	576	shrink-window-horizontally.....	523
set-match-data.....	680	shrink-window-if-larger-than-buffer.....	523
set-minibuffer-window.....	301	side effect.....	110
set-mouse-pixel-position.....	548	signal.....	128
set-mouse-position.....	547	signal-process.....	717
set-network-process-option.....	730	signaling errors.....	128
set-process-buffer.....	718	signals.....	715
set-process-coding-system.....	714	sigusr1 event.....	323
set-process-datagram-address.....	727	sigusr2 event.....	323
set-process-filter.....	720	sin.....	44
set-process-filter-multibyte.....	721	single-key-description.....	430
set-process-plist.....	714	sit-for.....	337
set-process-query-on-exit-flag.....	723	'site-init.el'.....	870
set-process-sentinel.....	722	'site-load.el'.....	870
set-register.....	634	site-run-file.....	814
set-right-margin.....	602	'site-start.el'.....	812
set-screen-height.....	540	size of frame.....	538
set-screen-width.....	540	size of window.....	520

skip-chars-backward.....	568	standard errors.....	891
skip-chars-forward.....	568	standard hooks.....	903
skip-syntax-backward.....	691	standard keymaps.....	899
skip-syntax-forward.....	691	standard regexps used in editing.....	683
skipping characters.....	567	standard-case-table.....	61
skipping comments.....	695	standard-category-table.....	697
sleep-for.....	338	standard-display-table.....	809
small-temporary-file-directory .....	460	standard-input.....	271
Snarf-documentation.....	428	standard-output.....	276
sort .....	77	standard-syntax-table.....	695
sort-columns.....	608	standard-translation-table-for-decode ...	647
sort-fields.....	608	standard-translation-table-for-encode ...	647
sort-fold-case.....	607	standards of coding style .....	856
sort-lines.....	608	start-process.....	710
sort-numeric-base.....	608	start-process-shell-command.....	711
sort-numeric-fields.....	608	startup of Emacs.....	812
sort-pages.....	608	'startup.el'.....	812
sort-paragraphs.....	608	staticpro, protection from GC.....	879
sort-regexp-fields.....	607	sticky text properties.....	625
sort-subr.....	605	stop points.....	246
sorting lists.....	77	stop-process.....	716
sorting text.....	605	stopping an infinite loop.....	239
sound .....	835	stopping on events.....	251
source breakpoints.....	251	store-match-data.....	680
space (ASCII character).....	10	store-substring.....	51
spaces, pixel specification.....	784	stream (for printing).....	271
spaces, specified height or width.....	783	stream (for reading).....	268
sparse keymap.....	348	string.....	48
SPC in minibuffer.....	281	string equality.....	52
special.....	387	string in keymap.....	358
special events.....	337	string input stream.....	269
special form descriptions.....	4	string length.....	87
special form evaluation .....	114	string quote.....	686
special forms .....	22	string search .....	661
special forms for control structures.....	119	string to character.....	54
special-display-buffer-names.....	509	string to number.....	55
special-display-frame-alist.....	510	string to object.....	271
special-display-function .....	510	string, number of bytes .....	641
special-display-p .....	510	string, writing a doc string .....	425
special-display-popup-frame.....	510	string-as-multibyte .....	643
special-display-regexp.....	510	string-as-unibyte .....	643
special-event-map .....	358	string-bytes .....	641
specify color.....	552	string-chars-consed .....	876
speedups.....	861	string-equal .....	52
splicing (with backquote).....	179	string-lessp .....	53
split-char .....	645	string-make-multibyte .....	642
split-height-threshold.....	509	string-make-unibyte .....	642
split-string .....	50	string-match .....	674
split-string-default-separators .....	51	string-or-null-p .....	48
split-window .....	498	string-to-char .....	54
split-window-horizontally .....	500	string-to-int .....	55
split-window-keep-point .....	500	string-to-multibyte .....	642
split-window-vertically .....	499	string-to-number .....	55
splitting windows.....	498	string-to-syntax .....	696
sqrt .....	45	string-width .....	760
stable sort.....	77	string<.....	52
standard buffer-local variables .....	895	string=.....	52
standard colors for character terminals .....	537	stringp .....	48

strings.....	47	syntax-after .....	696
strings with keyboard events.....	328	syntax-begin-function.....	693
strings, formatting them.....	56	syntax-class .....	696
<b>strings-consed.....</b>	876	syntax-ppss .....	693
subprocess .....	705	syntax-ppss-flush-cache .....	693
subr.....	160	syntax-ppss-toplevel-pos .....	694
<b>subr-arity.....</b>	161	syntax-table .....	690
subrp .....	161	syntax-table (text property) .....	690
subst-char-in-region.....	633	syntax-table-p .....	684
substitute-command-keys .....	429	system type and name.....	819
substitute-in-file-name .....	458	system-configuration.....	819
substitute-key-definition .....	363	system-key-alist.....	835
substituting keys in documentation .....	428	system-messages-locale.....	660
substring.....	48	system-name .....	820
substring-no-properties.....	49	system-time-locale .....	660
subtype of char-table.....	93	system-type .....	819
suggestions .....	1		
super characters.....	13		
<b>suppress-keymap.....</b>	364		
suspend (cf. no-redraw-on-reenter).....	739	<b>T</b>	
suspend evaluation .....	343	t .....	3
<b>suspend-emacs.....</b>	818	t input stream .....	269
suspend-hook .....	818	t output stream .....	272
suspend-resume-hook .....	819	tab (ASCII character) .....	10
suspending Emacs.....	817	tab deletion .....	588
<b>switch-to-buffer.....</b>	506	TAB in minibuffer .....	281
switch-to-buffer-other-window.....	507	tab-stop-list .....	613
switches on command line .....	816	tab-to-tab-stop .....	613
switching to a buffer.....	506	tab-width.....	807
<b>sxhash.....</b>	100	tabs stops for indentation .....	613
symbol.....	102	tag on run time stack .....	126
symbol components.....	102	tag, customization keyword .....	185
<b>symbol constituent.....</b>	685	tan .....	44
symbol equality .....	105	TCP .....	724
symbol evaluation .....	111	temacs .....	870
symbol function indirection .....	112	TEMP environment variable .....	460
symbol in keymap .....	359	temp-buffer-setup-hook .....	753
symbol name hashing .....	104	temp-buffer-show-function .....	752
symbol that evaluates to itself .....	135	temp-buffer-show-hook .....	753
symbol with constant value .....	135	temporary-file-directory .....	460
<b>symbol-file.....</b>	211	TERM environment variable .....	815
<b>symbol-function.....</b>	171	term-file-prefix .....	815
<b>symbol-name.....</b>	105	term-setup-hook .....	815
<b>symbol-plist.....</b>	108	Termcap .....	814
<b>symbol-value.....</b>	143	terminal frame .....	529
<b>symbolp.....</b>	102	terminal input .....	832
<b>symbols-consed.....</b>	876	terminal input modes .....	832
synchronous subprocess.....	707	terminal output .....	834
syntactic font lock .....	420	terminal screen .....	498
syntax class .....	684	terminal-coding-system .....	658
syntax descriptor .....	684	terminal-specific initialization .....	814
syntax error (Edebug) .....	262	termscript file .....	834
syntax flags .....	687	terpri .....	275
syntax for characters .....	10	test-completion .....	287
syntax table .....	684	testcover-mark-all .....	266
syntax table example .....	395	testcover-next-mark .....	266
syntax table internals.....	695	testcover-start .....	266
syntax tables in modes .....	386	testing types .....	27
		text .....	581

text deletion .....	587	transcendental functions .....	44
text files and binary files.....	658	transient-mark-mode .....	578
text insertion.....	585	translate-region.....	633
text near point .....	581	translation tables.....	647
text parsing .....	684	translation-table-for-input .....	648
text properties .....	615	transpose-regions .....	635
text properties in files .....	626	triple-click events.....	320
text properties in the mode line.....	409	true .....	3
text representations .....	640	true list .....	63
text-char-description .....	430	truename (of file).....	446
text-mode-abbrev-table.....	704	truncate .....	36
text-mode-map .....	901	truncate-lines .....	740
text-mode-syntax-table.....	695	truncate-partial-width-windows .....	740
text-properties-at .....	616	truncate-string-to-width .....	760
text-property-any .....	620	truth value .....	2
text-property-default-nonsticky .....	625	try-completion .....	285
text-property-not-all .....	620	tty-color-alist .....	554
textual scrolling .....	515	tty-color-approximate .....	554
thing-at-point .....	584	tty-color-clear .....	554
this-command .....	313	tty-color-define .....	554
this-command-keys .....	313	tty-color-translate .....	555
this-command-keys-vector .....	314	tty-erase-char .....	822
this-original-command .....	313	two's complement .....	32
three-step-help .....	433	type .....	8
throw .....	126	type (button property) .....	797
throw example .....	342	type checking .....	27
tiled windows .....	498	type checking internals .....	880
time-add .....	829	type predicates .....	27
time-less-p .....	828	type, defcustom keyword .....	191
time-subtract .....	828	type-of .....	29
time-to-day-in-year .....	829		
time-to-days .....	829		
timer .....	829		
timer-max-repeats .....	830	<b>U</b>	
timestamp of a mouse event .....	327	UDP .....	724
timing programs .....	861	umask .....	452
tips for writing Lisp .....	856	unbalanced parentheses .....	265
TMP environment variable .....	460	unbinding keys .....	368
TMPDIR environment variable .....	460	undefined .....	360
toggle-read-only .....	490	undefined in keymap .....	359
tool bar .....	378	undefined key .....	348
tool-bar-add-item .....	379	undo avoidance .....	633
tool-bar-add-item-from-menu .....	380	undo-ask-before-discard .....	599
tool-bar-border .....	381	undo-boundary .....	597
tool-bar-button-margin .....	380	undo-in-progress .....	598
tool-bar-button-relief .....	380	undo-limit .....	599
tool-bar-local-item-from-menu .....	380	undo-outer-limit .....	599
tool-bar-map .....	379	undo-strong-limit .....	599
tooltip .....	621	unexec .....	871
top-level .....	343	unhandled-file-name-directory .....	468
top-level form .....	201	unibyte text .....	640
tq-close .....	724	unibyte-char-to-multibyte .....	642
tq-create .....	724	unicode character escape .....	11
tq-enqueue .....	724	unintern .....	107
trace buffer .....	255	uninterned symbol .....	105
track-mouse .....	547	universal-argument .....	342
trailing codes .....	640	unless .....	121
transaction queue .....	723	unload-feature .....	211
		unload-feature-special-hooks .....	212

unloading packages	211	vector-cells-consed	876
unloading packages, preparing for	857	vectorp	92
<b>unlock-buffer</b>	443	verify-visited-file-modtime	488
unpacking	732	version number (in file name)	453
<b>unread-command-char</b>	336	version, customization keyword	186
<b>unread-command-events</b>	335	version-control	474
<b>unsafe-p</b>	174	vertical fractional scrolling	518
<b>unwind-protect</b>	133	vertical tab	10
unwinding	133	vertical-line prefix key	331
<b>up-list</b>	566	vertical-motion	564
upcase	59	vertical-scroll-bar	781
upcase-initials	59	vertical-scroll-bar prefix key	331
upcase-region	614	view part, model/view/controller	800
upcase-word	614	view-file	436
update-directory-autoloads	207	view-mode-map	902
update-file-autoloads	207	view-register	634
upper case	58	visible frame	545
upper case key sequence	331	visible-bell	810
<b>use-global-map</b>	356	visible-frame-list	542
<b>use-hard-newlines</b>	602	visited file	485
<b>use-local-map</b>	356	visited file mode	389
user identification	822	visited-file-modtime	489
user option	141	visiting files	434
user signals	323	void function	112
user-defined error	132	void function cell	171
<b>user-full-name</b>	823	void variable	138
<b>user-init-file</b>	814	void-function	171
<b>user-login-name</b>	823	void-text-area-pointer	550
<b>user-mail-address</b>	823	void-variable	138
<b>user-real-login-name</b>	823		
<b>user-real-uid</b>	823		
<b>user-uid</b>	823		
<b>user-variable-p</b>	141		
<b>user-variable-p example</b>	293		

**V**

value cell	102
value of expression	110
<b>values</b>	118
variable	135
variable aliases	157
variable definition	139
variable descriptions	6
variable limit error	137
variable with constant value	135
variable, buffer-local	147
<b>variable-documentation</b>	425
<b>variable-interactive</b>	141
variable-width spaces	783
variant coding system	648
<b>vc-mode</b>	406
<b>vc-prefix-map</b>	352
<b>vconcat</b>	92
<b>vector</b>	92
vector (type)	91
vector evaluation	111
vector length	87

**W**

waiting	337
waiting for command key input	336
<b>waiting-for-user-input-p</b>	723
<b>walk-windows</b>	505
<b>warn</b>	746
warning type	746
<b>warning-fill-prefix</b>	747
<b>warning-levels</b>	747
<b>warning-minimum-level</b>	748
<b>warning-minimum-log-level</b>	748
<b>warning-prefix-function</b>	747
<b>warning-series</b>	747
<b>warning-suppress-log-types</b>	748
<b>warning-suppress-types</b>	748
<b>warning-type-format</b>	747
warnings	745
<b>wheel-down event</b>	322
<b>wheel-up event</b>	322
<b>when</b>	121
<b>where-is-internal</b>	369
<b>while</b>	124
<b>while-no-input</b>	336
whitespace	10
whitespace character	685
<b>wholenump</b>	34
<b>widen</b>	570

widening .....	570	window-tree .....	525
window .....	497	window-vscroll .....	518
window (overlay property) .....	757	window-width .....	521
window configuration (Edebug) .....	257	windowp .....	498
window configurations .....	526	Windows file types .....	658
window excursions .....	569	windows, controlling precisely .....	505
window frame .....	529	with-case-table .....	61
window header line .....	409	with-current-buffer .....	483
window internals .....	886	with-local-quit .....	340
window layout in a frame .....	25	with-no-warnings .....	220
window layout, all frames .....	25	with-output-to-string .....	275
window manager, and frame parameters .....	536	with-output-to-temp-buffer .....	752
window ordering, cyclic .....	503	with-selected-window .....	502
window point .....	511	with-syntax-table .....	690
window point internals .....	887	with-temp-buffer .....	483
window position .....	511	with-temp-file .....	442
window resizing .....	522	with-temp-message .....	742
window size .....	520	with-timeout .....	830
window size, changing .....	522	word constituent .....	685
window splitting .....	498	word-search-backward .....	662
window start position .....	512	word-search-forward .....	662
window that satisfies a predicate .....	503	words-include-escapes .....	561
window top line .....	512	write-abbrev-file .....	702
window tree .....	525	write-char .....	275
window-at .....	524	write-contents-functions .....	439
window-body-height .....	521	write-file .....	438
window-buffer .....	505	write-file-functions .....	438
window-configuration-change-hook .....	528	write-region .....	441
window-configuration-frame .....	527	write-region-annotate-functions .....	626
window-configuration-p .....	527	writing a documentation string .....	425
window-current-scroll-bars .....	782	writing Emacs primitives .....	876
window-dedicated-p .....	511	writing to files .....	441
window-display-table .....	809	wrong-number-of-arguments .....	163
window-edges .....	521	wrong-type-argument .....	27
window-end .....	512		
window-frame .....	542		
window-fringes .....	777		
window-height .....	520	X	
window-hscroll .....	519	X Window System .....	811
window-inside-edges .....	521	x-alt-keysym .....	836
window-inside-pixel-edges .....	522	x-bitmap-file-path .....	767
window-line-height .....	514	x-close-connection .....	531
window-list .....	505	x-color-defined-p .....	553
window-live-p .....	501	x-color-values .....	554
window-margins .....	787	x-defined-colors .....	553
window-min-height .....	524	x-display-color-p .....	556
window-min-width .....	524	x-display-list .....	530
window-minibuffer-p .....	301	x-dnd-known-types .....	552
window-pixel-edges .....	522	x-dnd-test-function .....	552
window-point .....	512	x-dnd-types-alist .....	552
window-redisplay-end-trigger .....	528	x-family-fonts .....	774
window-scroll-bars .....	782	x-font-family-list .....	774
window-scroll-functions .....	527	x-get-cut-buffer .....	551
window-setup-hook .....	811	x-get-resource .....	555
window-size-change-functions .....	528	x-get-selection .....	551
window-size-fixed .....	524	x-hyper-keysym .....	836
window-start .....	512	x-list-fonts .....	773
window-system .....	811	x-meta-keysym .....	836
		x-open-connection .....	530

x-parse-geometry..... 540  
x-pointer-shape..... 550  
x-popup-dialog..... 549  
x-popup-menu..... 548  
x-resource-class..... 555  
x-resource-name..... 555  
x-select-enable-clipboard..... 552  
x-sensitive-text-pointer-shape ..... 550  
x-server-vendor..... 558  
x-server-version..... 558  
x-set-cut-buffer..... 551  
x-set-selection..... 550  
x-super-keysym..... 836  
X11 keysyms..... 835  
XBM..... 791  
XPM..... 792

**Y**

y-or-n-p..... 297  
y-or-n-p-with-timeout..... 298  
yank ..... 593  
yank suppression ..... 364  
yank-pop..... 593  
yank-undo-function ..... 594  
yes-or-no questions ..... 296  
yes-or-no-p ..... 298

**Z**

zerop ..... 34