37.2 Buffer Internals 353
37.3 Syntax Tables 354
37.4 Window Internals 355
37.5 Process Internals 356
37.6 Functions Written in C (primitives) 356
37.7 Garbage Collection 359
37.7.1 Pure Storage 361

Appendix A. Standard Buffer Local Variables 363

Appendix B. Some Constants and Variables which are Keymaps. 367

Appendix C. Hooks 369

32.1 Predicates on Processes . . . 313
32.2 Functions that Create Subprocesses . . . 313
32.2.1 Creating an Asynchronous Process . . . 314
32.2.2 Creating a Synchronous Process . . . 316
32.3 Process Information . . . 317
32.4 Sending Information to a Process . . . 319
32.4.1 Sending Input to a Process . . . 319
32.4.2 Sending Signals to Processes . . . 320
32.5 Receiving Information from a Process . . . 321
32.5.1 Process Filters . . . 322
32.5.2 Process Buffers . . . 323
32.5.3 Process Sentinels . . . 323
32.6 Deleting Processes . . . 324
32.7 Process Misc. . . . 325
32.8 Subprocess Functions for VMS . . . 325
32.9 TCP . . . 326

33. Operating System Interface . . . 327
33.1 Building Emacs . . . 327
33.2 When Emacs Starts Up . . . 328
33.3 OS Environment . . . 329
33.4 I/O . . . 332
33.5 Getting out of Emacs . . . 333

34. Miscellaneous Modes . . . 335
34.1 Telnet . . . 335
34.2 Terminal Emulator . . . 335

35. Functions and Variables NOT Documented yet: . . . 337
35.1 Abbreviations . . . 337

36. GNU Emacs Lisp for the Non-Hacker . . . 339
36.1 Lisp History . . . 339
36.2 Starting with Lisp . . . 339
36.3 Symbols . . . 340
36.4 Binding Variables Locally . . . 344
36.5 Writing Simple Functions . . . 346
36.6 Functions That Work on Text . . . 349
36.7 Writing Simple Commands . . . 349
36.8 Functions that Evaluate Several Expressions . . . 350
36.9 Real Functions Evaluate their Arguments . . . 352

37. GNU Emacs Internals . . . 353
37.1 Object Internals . . . 353

26.7 Temporary Displays . . . 253
26.8 Miscellaneous . . . 255

27. Positions . . . 257
27.1 Point . . . 257
27.2 Motion . . . 259
27.3 Goal Column . . . 264
27.4 Excursions . . . 265

28. Markers . . . 267
28.1 Predicates on Markers . . . 268
28.2 Functions which Create Markers . . . 268
28.3 Information from Markers . . . 270
28.4 Changing Markers . . . 270
28.5 The Mark . . . 271
28.6 Region . . . 273

29. Text . . . 275
29.1 Insertion . . . 275
29.2 Deletion . . . 278
29.3 The Kill Ring . . . 282
29.4 Tab Stops . . . 284
29.5 Auto Filling . . . 285
29.6 Sorting . . . 286
29.7 Indenting . . . 287
29.8 Case Changes . . . 289
29.9 Registers . . . 290

30. Searching and Matching . . . 293
30.1 Syntax of Regular Expressions . . . 293
30.2 Searching and Case . . . 296
30.3 Functions that Match Regular Expressions . . . 297
30.4 Replacements . . . 300
30.5 Match Data . . . 301
30.6 Searching for Strings . . . 303
30.7 Paragraphs . . . 304
30.8 Miscellaneous . . . 305

31. Syntax Tables . . . 307
31.1 Syntax Classes . . . 307
31.2 Syntax Table Predicates . . . 309
31.3 Syntax Table Functions . . . 309
31.4 Some Standard Syntax Tables . . . 312

32. Processes . . . 313

22.4.1 Information about Files . . . 205
22.4.2 Directories . . . 209
22.4.3 Modifying the File System . . . 210
22.5 File Names . . . 212
22.5.1 Functions that Expand Filenames . . . 215
22.5.2 Filename Completion . . . 216

23. Backups and Auto Saving . . . 217
23.1 Backup and Auto Save File Names . . . 217
23.2 How to make Backup files . . . 219
23.3 Editing Files on Remote Machines . . . 221

24. Buffers . . . 223
24.1 Information about Buffers . . . 223
24.2 Functions that Change Buffer Attributes . . . 225
24.2.1 The Buffer List . . . 226
24.3 Functions that Create and Delete Buffers . . . 228
24.4 Functions that Select Buffers . . . 229
24.5 Functions that Find Buffers that Already Exist . . . 231
24.6 Read Only Buffers . . . 231
24.7 Restrictions . . . 232
24.8 Undo . . . 233

25. Windows . . . 235
25.1 Predicates on Windows . . . 236
25.2 Splitting Windows . . . 236
25.3 Deleting Windows . . . 238
25.4 Selecting a Window . . . 239
25.5 Finding a window . . . 240
25.6 Buffer and Window . . . 241
25.7 Buffer Position . . . 241
25.8 Scrolling . . . 243
25.9 Screen Position . . . 245
25.10 Resizing Windows . . . 246
25.11 Window Configurations . . . 247

26. Emacs Display . . . 249
26.1 Screen Attributes . . . 249
26.2 Window Systems . . . 250
26.3 Truncation . . . 250
26.4 Sitting and Beeping . . . 251
26.5 Selective Display . . . 251
26.6 Overlay Arrow . . . 253

16.2 Output Streams . . . 131
16.2.1 Output Functions . . . 133
17. Commands . . . 137
17.1 Command Keys . . . 141
17.2 Aborting . . . 145
17.3 Prefix Command Arguments . . . 146
17.4 Recursive Editing Levels . . . 149
17.5 Disabling Commands . . . 150
17.6 Command History . . . 151
17.6.1 Electric Command History Mode . . . 152
17.7 Keyboard Macros . . . 153
18. Keymaps . . . 155
18.1 Keymap Entries . . . 156
18.2 Local and Global Keymaps . . . 157
18.3 Prefix Keys . . . 159
18.4 Changing Key Bindings . . . 161
18.5 Miscellaneous . . . 163
19. The Minibuffer . . . 167
19.1 Preliminaries . . . 167
19.2 Reading Text or Objects . . . 168
19.3 Completion . . . 172
19.4 Other uses . . . 178
20. Major and Minor Modes . . . 181
20.1 Major Modes . . . 181
20.1.1 Setting the Major Mode . . . 183
20.1.2 Information about Major Modes . . . 184
20.2 Minor Modes . . . 185
20.3 Mode Line Format . . . 186
20.3.1 Example Mode Lines . . . 189
20.4 Hooks . . . 190
21. Documentation . . . 193
21.1 Help Functions . . . 197
22. Files . . . 199
22.1 Finding and visiting files . . . 199
22.1.1 Reverting . . . 202
22.2 Writing to Files . . . 202
22.3 File Modtime . . . 204
22.3.1 File Locks . . . 205
22.4 File System Interface . . . 205

8.1 Symbol Components . . . 65
8.2 Uses of Symbols . . . 66
8.3 Predicates on Symbols . . . 66
8.4 Creating and Interning Symbols . . . 66
8.5 Property Lists . . . 69

9. Variables . . . 71
9.1 Scope and Extent . . . 71
9.2 Global Variables . . . 73
9.3 Binding Local Variables . . . 76
9.4 Referencing and Changing the Value of Symbols . . . 77
9.5 Buffer Local Variables . . . 79

10. Functions . . . 83
10.1 Lambda Expressions . . . 84
10.2 Function Declarations . . . 87
10.3 Manipulating Symbols' Function Cell . . . 88
10.4 Other Function Related Functions . . . 89

11. Macros . . . 91
11.1 Predicates for Macros . . . 93
11.2 Functions for Macros . . . 93
11.3 Backquote . . . 94

12. Control Structures . . . 97
12.1 Function Invocation . . . 97
12.2 Mapping Functions . . . 98
12.3 Sequencing . . . 99
12.4 Iteration . . . 100
12.5 Conditionals . . . 101
12.6 Non-local Exits . . . 103
12.7 Errors . . . 107
12.7.1 Error Conditions . . . 110

13. Loading . . . 113
13.1 Autoload . . . 114
13.2 Features . . . 115

14. Byte Compilation . . . 117

15. Debugging . . . 121
15.1 Debugger Mode Commands . . . 125

16. Input/Output Streams . . . 127
16.1 Input Streams . . . 127
16.1.1 Input Functions . . . 129

2.16.1 Type Predicates 20
2.16.2 Equality Predicates 21

3. Evaluation 23
3.1 Eval 23
3.2 Forms 24
3.2.1 Self-Evaluating Forms 25
3.2.2 Symbol Forms 25
3.2.3 List Forms 26
3.2.3.1 Function Calls 27
3.2.3.2 Macro Calls 27
3.2.3.3 Primitive Function Calls 28
3.2.3.4 28
3.2.3.5 Definitions and Declarations 29
3.3 Identity and Quotes 30

4. Numbers 33
4.1 Predicates on Numbers 33
4.2 Comparison on Numbers 34
4.3 Arithmetic Operations 35
4.4 Bitwise Operations on Integers 37
4.5 Random Numbers 39

5. Strings and Characters 41
5.1 Predicates for Strings 41
5.2 Creating Strings 41
5.3 Character and String Conversion 44
5.4 Character Case 46

6. Lists 49
6.1 Predicates 49
6.2 Building Cons Cells and Lists 50
6.3 Accessing Elements of Lists 51
6.4 Alteration of List Structure 53
6.5 Using Lists as Sets 55
6.6 Association Lists 56

7. Sequences, Arrays, and Vectors 59
7.1 Sequences 59
7.2 Arrays 61
7.2.1 Predicates for Arrays 61
7.2.2 Accessing and Changing Elements of an Array 61
7.3 Vectors 62

8. Symbols 65

Table of Contents

1. Introduction . . . 3
1.1 Lisp History . . . 3
1.2 Conventions . . . 3
1.3 The Lisp Reader and Printer . . . 3
1.4 Nil, False, and the Empty List . . . 4
1.5 Evaluation, Expansion, and Equivalence . . . 4
1.6 Printing . . . 5
1.7 Errors . . . 5
1.8 Inserting Text . . . 5
1.9 The Format of Descriptions . . . 6
1.9.1 A Sample Function Description . . . 6
1.9.2 A Sample Variable, Option, or Constant Description . . . 7

2. Types of Lisp Objects . . . 9
2.1 Print Representation and Read Syntax . . . 9
2.2 Number Type . . . 10
2.3 Character Type . . . 10
2.4 Sequence Type . . . 11
2.5 List Type . . . 12
2.6 Array Type . . . 12
2.6.1 String Type . . . 13
2.6.2 Vector Type . . . 14
2.7 Symbol Type . . . 15
2.8 Primitive Function Type . . . 16
2.9 Buffer Type . . . 16
2.10 Marker Type . . . 17
2.11 Window Type . . . 17
2.12 Window Configuration Type . . . 17
2.13 Process Type . . . 18
2.14 Stream Type . . . 18
2.15 Derived Types . . . 18
2.15.1 Association List Type . . . 18
2.15.2 Lisp Function Type . . . 18
2.15.3 Lisp Macro Type . . . 19
2.15.4 Autoload Type . . . 19
2.15.5 Byte Code Type . . . 19
2.15.6 Syntax Table Type . . . 19
2.15.7 Keymap Type . . . 20
2.16 Predicates . . . 20

Appendix C. Hooks 369

Summary Table of Contents

1. Introduction . . . 3
2. Types of Lisp Objects . . . 9
3. Evaluation . . . 23
4. Numbers . . . 33
5. Strings and Characters . . . 41
6. Lists . . . 49
7. Sequences, Arrays, and Vectors . . . 59
8. Symbols . . . 65
9. Variables . . . 71
10. Functions . . . 83
11. Macros . . . 91
12. Control Structures . . . 97
13. Loading . . . 113
14. Byte Compilation . . . 117
15. Debugging . . . 121
16. Input/Output Streams . . . 127
17. Commands . . . 137
18. Keymaps . . . 155
19. The Minibuffer . . . 167
20. Major and Minor Modes . . . 181
21. Documentation . . . 193
22. Files . . . 199
23. Backups and Auto Saving . . . 217
24. Buffers . . . 223
25. Windows . . . 235
26. Emacs Display . . . 249
27. Positions . . . 257
28. Markers . . . 267
29. Text . . . 275
30. Searching and Matching . . . 293
31. Syntax Tables . . . 307
32. Processes . . . 313
33. Operating System Interface . . . 327
34. Miscellaneous Modes . . . 335
35. Functions and Variables NOT Documented yet: . . . 337
36. GNU Emacs Lisp for the Non-Hacker . . . 339
37. GNU Emacs Internals . . . 353
Appendix A. Standard Buffer Local Variables . . . 363
Appendix B. Some Constants and Variables which are Keymaps. . . . 367

view-hook

write-file-hooks

x-process-mouse-hook

lisp-mode-hook
m2-mode-hook
mail-mode-hook
mail-setup-hook
medit-mode-hook
mh-compose-letter-hook
mh-folder-mode-hook
mh-letter-mode-hook
mim-mode-hook
news-mode-hook
news-reply-mode-hook
news-setup-hook
nroff-mode-hook
outline-mode-hook
plain-TeX-mode-hook
prolog-mode-hook
protect-innocence-hook
rmail-edit-mode-hook
rmail-mode-hook
rmail-summary-mode-hook
scheme-indent-hook
scheme-mode-hook
scribe-mode-hook
shell-mode-hook
shell-set-directory-error-hook
suspend-hook
suspend-resume-hook
temp-buffer-show-hook
term-setup-hook
terminal-mode-hook
terminal-mode-break-hook
TeX-mode-hook
text-mode-hook
vi-mode-hook

Appendix C. Hooks

The following is a list of hooks available with the distributed 18.52 version of GNU Emacs. Some of these hooks are called with `run-hooks` and can be a list of functions. Others are not called with `run-hooks` and may or may not allow a list of functions. For example, the `suspend-hook` can only reference a single function.

(!! move to appendix. xref to where documented or document it)

`auto-fill-hook`
`blink-paren-hook`
`c-mode-hook`
`command-history-hook`
`comment-indent-hook`
`define-hooked-global-abbrev`
`define-hooked-local-abbrev`
`dired-mode-hook`
`disabled-command-hook`
`edit-picture-hook`
`edit-picture-hook`
`electric-buffer-menu-mode-hook`
`electric-command-history-hook`
`electric-help-mode-hook`
`emacs-lisp-mode-hook`
`find-file-hooks`
`find-file-not-found-hooks`
`fortran-comment-hook`
`fortran-mode-hook`
`ftp-setup-write-file-hooks`
`ftp-write-file-hook`
`indent-mim-hook`
`LaTeX-mode-hook`
`ledit-mode-hook`
`lisp-indent-hook`
`lisp-interaction-mode-hook`

`occur-mode-map`
: The value of this variable is a local keymap used in occur mode.

`text-mode-map`
: The value of this variable is a sparse keymap used by text mode.

`view-mode-map`
: The value of this variable is a full keymap used by view mode.

`debugger-mode-map`
: The value of this variable is a full keymap used by Debugger mode.

`command-history-map`
: The value of this variable is a full keymap used by Command History mode.

`doctor-mode-map`
: The value of this variable is a sparse keymap used by doctor mode.

`electric-buffer-menu-mode-map`
: The value of this variable is a full keymap used by Electric buffer menu mode.

`electric-history-map`
: The value of this variable is a full keymap used by electric command history mode.

`fundamental-mode-map`
: The value of this variable is the local keymap for fundamental mode. It is empty unless changed by the user.

`mode-specific-map`
: The value of this variable is the keymap for characters following `C-c`.

Appendix B. Some Constants and Variables which are Keymap

The following symbols are used as the names for various keymaps. Some of these exist when Emacs is first started, others are only loaded when their respective mode is used. This is not an exhaustive list.

Almost all of these maps are used as local maps. Indeed, of the modes that presently exist, only "vip mode" and "terminal emulator mode" ever change the global keymap.

`Helper-help-map`
: The value of this variable is a full keymap used by the help utility package. It has the same keymap in its value cell and in its function cell.

`Buffer-menu-mode-map`
: The value of this variable is a full keymap used by Buffer mode.

`Info-edit-map`
: The value of this variable is a sparse keymap used by the 'e' command of Info.

`Info-mode-map`
: The value of this variable is a sparse keymap containing Info commands.

`c-mode-map`
: The value of this variable is a sparse keymap used in C mode as a local map.

`ctl-x-4-map`
: The value of this variable is a sparse keymap for subcommands of the prefix `C-x 4`.

`ctl-x-map`
: The value of this variable is a default keymap for `C-x` commands.

`dired-mode-map`
: The value of this variable is a full keymap for dired-mode buffers.

`edit-abbrevs-map`
: The value of this variable is a sparse keymap used in edit-abbrevs.

`edit-tab-stops-map`
: The value of this variable is a sparse keymap used in edit-tab-stops.

`emacs-lisp-mode-map`
: The value of this variable is a sparse keymap used in Emacs Lisp Mode.

`function-keymap`
: This variable contains the keymap for the definitions of keypad and function keys. If there are none, then it contains an empty sparse keymap.

`mouse-map`
: The value of this variable is a sparse keymap for mouse commands from the X window system.

default-directory Variable

The value of this buffer-local variable is the the default directory for the current buffer. This is always a string ending with a slash on unix system.

```
default-directory
=> "/user/lewis/manual/"
```

require-final-newline User Option

This global variable determines if files may be written out that do `not` end with a newline. If it is `t`, the Emacs silently puts a newline at the end whenever a file being saved does not already end in one. If it is non-`nil`, but not `t`, then Emacs asks the user whether to add a newline in each such case.

If it is `nil`, then Emacs doesn't add newlines at all.

`buffer-auto-save-file-name`
: see chapter 37 [buffer-auto-save-file-name], page 364

`buffer-saved-size`
: see chapter 37 [buffer-saved-size], page 364

`buffer-backed-up`
: see chapter 37 [buffer-backed-up], page 364

`default-directory`
: see chapter 37 [default-directory], page 365

`buffer-file-name`
: see section 24.1 [buffer-file-name], page 224

`mark-ring`
: see section 28.5 [mark-ring], page 273

`paragraph-start`
: see section 30.7 [paragraph-start], page 304

`paragraph-separate`
: see section 30.7 [paragraph-separate], page 304

`require-final-newline`
: see chapter 37 [require-final-newline], page 365

local-abbrev-table Variable

The value of this global variable is the (mode-specific) abbreviation table of the current buffer.

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/#file.texinfo#"
```

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.

buffer-backed-up Variable

This buffer-local variable determines if this buffer's file has been backed up. If it is non-`nil`, then the backup file has been written. Backing up is done just before the first time the file is saved.

Appendix A. Standard Buffer Local Variables

The table below shows all of the buffer-local variables for Emacs version 18 with the common packages loaded.

`fill-column`
: see section 29.5 [fill-column], page 286

`truncate-lines`
: see section 26.3 [truncate-lines], page 250

`ctl-arrow`
: see section 26.8 [ctl-arrow], page 255

`selective-display`
: see section 26.5 [selective-display], page 251

`selective-display-ellipses`
: see section 26.5 [selective-display-ellipses], page 252

`minor-modes`
: see section 20.2 [Minor Modes], page 185

`overwrite-mode`
: see section 29.1 [overwrite-mode], page 275

`abbrev-mode`
: see <undefined> [Abbrev Mode], page <undefined>

`tab-width`
: see section 29.4 [tab-width], page 285

`auto-fill-hook`
: see section 29.5 [auto-fill-hook], page 286

`case-fold-search`
: see section 30.8 [case-fold-search], page 305

`left-margin`
: see <undefined> [Left Margin], page <undefined>

`local-abbrev-table`
: see chapter 37 [local-abbrev-table], page 364

`mode-name`
: see section 20.1.2 [mode-name], page 185

`major-mode`
: see section 20.1.2 [major-mode], page 184

`buffer-read-only`
: see section 24.6 [buffer-read-only], page 231

This flag is `t` while loading all of the basic functions for building Emacs initially (allowing those functions to be sharable and non-collectable). It is set to `nil` when Emacs is saved out as a runnable program.

There is a finite amount of space allocated to pure storage, and this amount is based upon the number of functions there are that are normally loaded when building Emacs. You should not change this flag in a running Emacs.

37.7.1 Pure Storage

There are two types of storage in GNU Emacs Lisp for user-created Lisp objects: *normal storage* and *pure storage*. Normal storage is where all the new data which is created by any session with Emacs is kept. When a program conses up a list or the user defines a new function (or loads a package), then that is placed in normal storage.

If normal storage runs low, then Emacs requests the Operating System to allocate more memory in blocks of 1k bytes. Each block is allocated for one type only, meaning that symbols, conses, vectors, etc. are segregated in distinct blocks in memory.

After a certain amount of storage has been allocated (determined by the user variable `gc-cons-threshold`), the garbage collector is called to collect all storage which has been used and abandoned.

Pure storage is unique in that it is not expandable, it will not be collected by the garbage collector, and it is also sharable — meaning that if two people are running Emacs, the operating system may choose to let them share the exact same memory for all the sharable portions of Emacs.

In essence, pure storage is used when initially building Emacs from the sources, loading all of the normal files which everyone will want. Normal storage is used during sessions with Emacs.

purecopy *object* Function

This function makes a copy of *object* in pure storage are returns it. It copies strings by simply making a new string with the same characters in pure storage. It recursively copies the contents of vectors and cons cells.

It does not make copies of symbols, or any other objects, but just returns them unchanged. It signals an error if asked to copy markers. This function is used only while Emacs is being built and dumped.

pure-bytes-used Variable

The value of this variable is the number of bytes of pure storage allocated so far. During a session, this number will be very close to the total amount of pure storage that exists.

purify-flag Variable

This variable determines if `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.

stack.), and marking those as being in use. Any objects which are accessable through them are also marked as being in use, and all the rest are must therefore be garbage.

Unused cons cells are simply strung together onto a *free list* for future allocation. Strings are first "compacted", and then all unused string space is made available to the string creation functions.

CN: Unlike other Lisps, the garbage collector is not called when storage is exhausted. Instead, GNU Emacs Lisp simply requests the Operating System to allocate more storage, and processing continues until `gc-cons-threshold` bytes have been used

---

garbage-collect Command

This function reclaims storage used by Lisp objects that are no longer needed. It returns information on the amount of space in use. Garbage collection happens automatically if you use more than `gc-cons-threshold` bytes of Lisp data since the previous garbage collection.

This function returns a list containing the following information:

```
((used-conses . free-conses)
 (used-syms . free-syms)
 (used-markers . free-markers)
 used-string-chars used-vector-slots)
```

```
(garbage-collect)
=> ((3435 . 2332) (1688 . 0) (57 . 417) 24510 3839)
```

gc-cons-threshold User Option

The value of this variable is the number of bytes of storage that may be used after one garbage collection before another one is automatically called. Storage is "used" by *consing* (a cons cell is eight bytes), creating strings (one byte per character plus a few bytes of overhead), adding text to a buffer, etc.

This variable may be set to a value above 100,000 to reduce the frequency of garbage collections. Setting it to a value below 100,000 will only have effect until the subsequent garbage collection, at which time `garbage-collect` will set it back to 100,000. (??)

```
  CHECK_WINDOW (window, 2);
  xcoord = Fcar (coordinate);
  ycoord = Fcar (Fcdr (coordinate));
  CHECK_NUMBER (xcoord, 0);
  CHECK_NUMBER (ycoord, 1);
  if ((XINT (xcoord) < XINT (XWINDOW (window)->left)) ||
      (XINT (xcoord) >= (XINT (XWINDOW (window)->left) +
                         XINT (XWINDOW (window)->width))))
    {
      return Qnil;
    }
  XFASTINT (xcoord) -= XFASTINT (XWINDOW (window)->left);
  if (XINT (ycoord) == (screen_height - 1))
    return Qnil;
  if ((XINT (ycoord) < XINT (XWINDOW (window)->top)) ||
      (XINT (ycoord) >= (XINT (XWINDOW (window)->top) +
                         XINT (XWINDOW (window)->height)) - 1))
    {
      return Qnil;
    }
  XFASTINT (ycoord) -= XFASTINT (XWINDOW (window)->top);
  return (Fcons (xcoord, Fcons (ycoord, Qnil)));
}
```

There are similar equivalents for `defvar` and `defconst`, as well as a few others that have no equivalent in the Lisp interpreter.

Note that you cannot directly call functions defined in Lisp as, for example, `Fcons` is called above. You must create the appropriate Lisp form, protect everything from garbage collection, and `Feval` the form, as was done in `For` above.

'`eval.c`' is a very good file to look through for examples; '`lisp.h`' contains the definitions for some important macros and functions.

37.7 Garbage Collection

All functions which build new structures (be they lists, strings, buffers, etc.) require storage space for those structures. As it is quite common to use some storage for a while, and then release it (e.g., by killing a buffer or deleting the last pointer to an object), a *Garbage Collector* is provided to reclaim this abandoned storage.

This Garbage Collector operates by scanning over all objects which are accessable by the user (e.g., all symbols, their values and associated function definitions, and any data presently on the

number of required arguments. In this case, no arguments are required.

The fifth argument is the maximum number of arguments that can be provided. This is a special form because this number is the macro UNEVALLED, indicating that the arguments are not to be evaluated. A function with the equivalent of an &rest argument would have the macro MANY in this position. This argument must be one of these macros or a number at least as large as the fourth argument.

The sixth argument is an interactive specification exactly like the one provided in Lisp. In this case it is 0 (a null pointer), indicating that this function cannot be called interactively. A value of "" indicates an interactive function not taking arguments.

The last argument is the documentation string.

A list of arguments must be provided, and their types (all Lisp objects) must be declared.

If you are modifying 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 a line of the form

```
defsubr (&Sname);
```

If the file doesn't have this function, or you have created a new file, add a `syms_of_`*filename*, e. g. `syms_of_eval`, and find the spot in '`emacs.c`' where all of these functions are called. Add a call to your symbol initialization function there. This makes all the subroutines (primitives) available from Lisp.

Here is another function, with more complicated arguments. This comes from the code for the X window system, 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 POSITIONS (a list, (SCREEN-X SCREEN-Y)) is in WINDOW.\n\
Returned value is list of positions expressed\n\
relative to window upper left corner.")
  (coordinate, window)
     register Lisp_Object coordinate, window;
{
  register Lisp_Object xcoord, ycoord;

  if (!LISTP  (coordinate)) wrong_type_argument (Qlistp, coordinate);
```

```
DEFUN ("or", For, Sor, 0, UNEVALLED, 0,
  "Eval args until one of them yields non-NIL, then return that value.\n\
The remaining args are not evalled at all.\n\
If all args return NIL, return NIL.")
  (args)
     Lisp_Object args;
{
  register Lisp_Object val;
  Lisp_Object args_left;
  struct gcpro gcpro1;

  if (NULL(args))
    return Qnil;

  args_left = args;
  GCPRO1 (args_left);

  do
    {
      val = Feval (Fcar (args_left));
      if (!NULL (val))
        break;
      args_left = Fcdr (args_left);
    }
  while (!NULL(args_left));

  UNGCPRO;
  return val;
}
```

Here is a precise explanation of the arguments to the `DEFUN` macro.

The first argument is the name of the function in Lisp; it will be named '`or`'.

The second argument 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, if your C code wishes to call this function, it will call For(...). Remember that the arguments must be Lisp_Objects; various macros and functions for creating Lisp_Objects are provided in the file lisp.h.

The third argument is the name of the C variable representing the Lisp primitive that this function codes. This name is by convention "S" prepended to the name, in the same manner that the function name is created.

The fourth argument is the minimum number of arguments that must be provided; i. e. the

37.5 Process Internals

The fields of a process are:

`name` A string: the name used when creating the process, or a variant of it. The name is normally created from the name of the program which it is running. (true??)

`command` A list: the command arguments that this process was created with.

`filter` A function (or a symbol which names it): used to accept output from the process instead of a buffer.

`sentinel` A function: to be called whenever the process receives a signal.

`buffer` A buffer: where standard output is directed to if filter does not exist.

`pid` An integer: the Unix process ID.

`command_channel_p`
A flag: Non-`nil` if this is really a command channel instead of a process??

`childp` A flag: Non-`nil` if this is really a child process.

`flags` A symbol: representing the state of the process, `run`, `stop`, `closed`, etc.

`reason` A number: the Unix signal that the process received that caused the process to stop (The process is not necessarily dead). If the process has died, and was not killed by a signal, then this is the code the process exited with.

`mark` A Marker: set to end of last output from this process inserted into the buffer. Normally this will be the end of the buffer.

`kill_without_query`
A flag: Non-`nil` means kill silently if Emacs is exited.

37.6 Functions Written in C (primitives)

Certain functions, and all special forms, are written in C. A convenient interface is provided via a set of macros. The only way to really understand how to write new C code is to read the source; however, some information will be provided here.

An example of a special form (an ordinary function would have the same general appearance) is the definition of `or`, from '`eval.c`'.

```
/* NOTE!!! Every function that can call EVAL must protect its args
 and temporaries from garbage collection while it needs them.
 The definition of 'For' shows what you have to do.  */
```

`11` comment-start

`12` comment-end

37.4 Window Internals

Windows have the following accessable attributes:

`height` The height of the window, measured in lines. All windows (save the minibuffer window) have a minimum height of two lines, one of which is the mode line.

`width` The width of the window, measured in columns. There is no limit on the width of a window, although a width of less than 2 does not allow for any characters at all to be displayed (as one line is devoted to the horizontal division line).

`buffer` The buffer which the window is displaying. This may change often during the life of the window.

`start` The position in the buffer which is the first character to be displayed in the buffer. This is always located in the upper left corner (location `(0,0)`). There is no restriction on which character in a buffer this is, although it is common for it to be the first character on a line.

`pointm` This is the point of the current buffer when this window is selected, and when it is not selected, it retains its previous value. When reselected, this once again becomes the point for the current buffer (assuming that the buffer associated with the window has not changed).

This allows Emacs to maintain different points in different windows, even when they display the same buffer. (This variable is only useful when there are multiple windows displaying the same buffer.)

`left` This is the left-hand edge of the window, measured in columns. (The left-most 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.)

`next` This is the window that will be returned when `next-window` is called.

`prev` This is the window that will be returned when `previous-window` is called.

`force-start`
This determines if the next redisplay may move the start of display.??

`hscroll` This is the number of columns that the display in the window is scrolled horizontally to the left. Normally this is 0.

`use-time` This is the last time that the window was selected. This field is used by `get-lru-window`.

`undodata` This variable contains records which tell Emacs how it can undo the last set of changes to the buffer See section 24.8 [undo], page 233.

`syntax_table_v`
This variable contains the syntax table for the buffer. See section 37.3 [Syntax Tables], page 354

`markers` This variable contains the list of all markers that point into the buffer. Whenever the contents of the buffer is modified, all of these markers must be updated. See chapter 28 [Markers], page 267

`backup_up`
This variable is a flag which tells if the visited file has been backed up.

`mark` This variable contains the mark for the buffer. The mark is a marker, hence it is also included on the list `markers`.

`local_var_alist`
This variable contains the association list containing all of the local variables and their associated values. A copy of this list is returned by the function `buffer-local-variables`.

`mode_line_format`
???

37.3 Syntax Tables

The actual interpretations of syntax table entries is described here.

`Integer`	Class
`0`	whitespace
`1`	punctuation
`2`	word
`3`	symbol
`4`	Open paren
`5`	Close paren
`6`	Expression prefix
`7`	string quote
`8`	
`9`	character quote
`10`	

37. GNU Emacs Internals

This chapter describes many internal aspects of GNU Emacs that might only be interesting to a few programmers.

37.1 Object Internals

GNU Emacs Lisp manipulates numerous different types. The actual data is stored in a heap and the only access that programs have to the data is through pointers. Pointers are thirty-two bits wide in most implementations. Twenty-four to Twenty-six bits are used to indicate the object in question, and the remaining six to eight bits make up a tag which identifies the object's type.

Because all access to data is through tagged pointers, it is always possible to determine the type of any object. This allows variables to be untyped, and the values assigned to them to be changed without regard to type. Function arguments also can be of any type; 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 2.16 [Predicates], page 20).

37.2 Buffer Internals

Buffers have a set of variables associated with them which are not directly accessable by the programmer. The names given to them are not usable by the programmer in any way, but there are often functions which can access and change their values. They are (as of Emacs 18):

`name` The buffer name is a string which names the buffer. It is guaranteed to be unique. See section 24.1 [buffer-name], page 224

`save_modified`
This variable contains the time when the buffer was last saved. xref??

`modtime` This variable contains modification time of the visited file. It is set when the file is written or read. Every time the buffer is written to the file, this variable is compared to the modification of the file. See section 22.3 [verify-visited-file-modtime], page 204

`auto_save_modified`
This variable contains the time when the buffer was last auto-saved.

`last_window_start`
This variable contains the position in the buffer at which the display started the last time the buffer was displayed in a window.

[end]

36.9 Real Functions Evaluate their Arguments

...

What if I changed them `both` to be `m1`? Disaster!

When I was young, I called my parents "mommy" and "daddy". Now I call them "mom" and "dad". It's no problem at all. Now if I called them `both` "dad", come Mother's day I'd be in real trouble.

word; something we had no command to do before!

Is it clear how this function works? It moves point forward to the end of the word with `(forward-word 1)`. Then it moves backward one character with `(backward-char 1)` so that point is positioned on the last letter of the word. Finally it capitalizes that word, starting with the letter point is on.

The effect is the same as typing `M-f C-b M-c`. This is no surprise, because (as `C-h c` will confirm for you) `M-f` runs `forward-word`, `C-b` runs `backward-char` and `M-c` runs `capitalize-word`.

The second thing we want to do is to make it possible to execute our new function via a key sequence. To do so, we need to introduce a new concept: that of function which are *interactive*.

All functions can be executed by typing their names in an expression and executing it (with `eval-last-expression` or several similar commands). But only some functions can be executed by defining key sequences for them. This is what we mean by interactive. Interactive functions are also called *commands*.

Whenever you define a function, you can make it interactive if you want to trouble to do so. Usually we do so only when we can see that users would want to use function interactively. For example, the function `forward-word` was made interactive because users want to have a key to move forward a word. But the function `+` was not made interactive, because it was not useful to make it so.

The way we make a function interactive is to put the expression `(interactive)` in its definition as the first expression to be evaluated. Here is our function made interactive, followed by an expression that will bind it to the key sequence `C-x !`:

```
(defun capitalize-last-letter ()
  (interactive)
  (forward-word 1)
  (backward-char 1)
  (capitalize-word 1))
=> capitalize-last-letter

(define-key global-map "\C-x!" 'capitalize-last-letter)
=> capitalize-last-letter
```

Evaluate these expressions and then try out the function using its new key binding.

`backward-2` is not a very useful function, and we're quite happy to let it disappear at the end of this editing session. In general, though, we want to keep the functions we write around in a file for future use.

Let's create a new file which we'll keep all of our soon-to-be-created functions in. Choose a name with an extension of '`.el`' (I'll choose the name '`myfuns.el`'). Emacs recognizes the '`.el`' ending and automatically enters Emacs-Lisp mode.

Now type in the definition for another very simple function:

```
(defun forward-word-2 ()
  (forward-word 2))
```

The command `LFD` is not available here, because it is a special feature of Lisp Interaction mode. The recommended way to evaluate this expression is to position point anywhere within the expression (or even at the end of its last line) and type `C-M-x` (`eval-defun`).

So, position the point somewhere within the function definition and type `C-M-x`. `forward-word-2` should appear in the echo area. Then position the point at the top of the buffer and execute the function `forward-word-2` using `M-ESC (forward-word-2) RET`, and you will see that the function does its job.

36.8 Functions that Evaluate Several Expressions

Like `backward-2`, `forward-word-2` is not a very useful function. It doesn't do anything that we couldn't do easier with other functions, and it can't be executed by a key sequence.

To write a function that does something new, we really need to combine several functions. Just as in the example using `let`, a function can evaluate several expressions. Here is a function that has three expressions in its body. The expressions will be evaluated in the order they appear.

```
(defun capitalize-last-letter ()
  (forward-word 1)
  (backward-char 1)
  (capitalize-word 1))
```

Type this definition into your buffer and evaluate it. Now position the point in front of some word in the buffer and call this function with `M-ESC`. It should capitalize the last letter of that

36.6 Functions That Work on Text

Because GNU Emacs is a text editor, not just a calculator, functions that change the text in an editor buffer are very important.

But you can't conveniently try out these functions in the `*scratch*` buffer, the way you can try out arithmetic, because these functions typically operate on the current buffer. If you use the `*scratch*` buffer to call them, they will always operate on the `*scratch*` buffer. (And if they do so while `LFD` is trying to read and evaluate expressions from that buffer, `LFD` will get confused!)

What you would really like to do is evaluate an expression, operating on the buffer of your choice.

To do this, use the command `M-ESC (eval-expression)`. This command is available in all buffers. It reads an expression in the minibuffer, evaluates it, and prints the returned value in the echo area. If the expression operates on the current buffer, it will operate on whatever buffer you were in when you typed the `M-ESC` command. Try it: Evaluate the expression `(+ 1 2)` by typing: `M-ESC (+ 1 2) RET`.

36.7 Writing Simple Commands

You already know that you can execute commands in Emacs by typing pre-defined key sequences. `C-b`, for example, executes the command `backward-char`, and moves the point backward one character. As it turns out, `backward-char` is just another function. Therefore, we can execute the command `backward-char` by evaluating the function.

Move the point backwards two characters by typing: `M-ESC (backward-char 2) RET`.

Now we have all the building blocks we need to create our own commands. To do so, we just write our own functions which we can then evaluate. Here's the definition of a very simple function:

```
(defun backward-2 ()
  (backward-char 2))
=> backward-2
```

Evaluating this function in the minibuffer (`M-ESC (backward-2) RET`) will move the point in the current buffer (`not` the minibuffer) backwards two characters and return the value `nil`, which will be printed in the minibuffer.

In the body of `circle` we multiplied `radius` by itself: we squared `radius`. We already have a function `square` to square a number. We can, if we wish, use it in defining `circle`.

Here is an alternate definition of `circle` that uses `square`:

```
(defun circle (radius)
  (* 3 (square radius)))
=> circle

(circle 10)
=> 300
(circle (+ 3 4))
=> 147
```

Which definition is better? Which ever one you find clearest and easiest to write. There are other considerations that we might like to take into account later on (which is fastest, which takes up more memory), but right now the most important thing is writing one that's clear.

The parameter of `circle` is `radius`, and the parameter of `square` is `length`. Would it have made any difference if we'd used different names for the parameters? Would it have made any difference if we'd used the `same` name for both parameters?

No. Each function has its own idea of what its parameters have for values, and they don't interfere. Just as in `let` statements, each bound variable is protected from all other variables with the same name. Each function "has its own sheet of paper" to work on.

Here are our two functions, each redefined to use the parameter name `x`. This change has no effect on what the functions do.

```
(defun circle (x)
  (* 3 (square x)))
=> circle
(defun square (x)
  (* x x))
=> square
(square 10)
=> 100
(circle 10)
=> 300
```

```
=> square
```

Because we have no idea of what the argument to `square` is going to be (it certainly won't always be 10), we need a name for it in order to talk about it. In Lisp, we use a variable for this; here, the variable is called `length`, but any name would do equally well. The variables used for the function's arguments are called the *parameters* of the function. In a `defun` construct, the parameters always appear in parentheses following the function name. (Some functions need no parameters; for them, you must write just a pair of parentheses with no variables in between.)

When the function is called, the parameter variables receive local bindings just like the variables in a `let`. They also receive local values, which are the function's arguments. Thus, when `square` is called, the variable `length` gets a local binding and its local value is the number to be squared.

`(* length length)` is the calculation that the function will perform. We call such a calculation the *body* of the function. (We can also speak of the body of a `let`.) The value returned by a function is always the result of evaluating the body.

In the first example (`(square 10)`), the parameter `length` becomes bound to the value 10. In the body of the function `square`, `length` is multiplied by itself and `square` returns the value 100.

In the second example ((square (+ 3 4))), exactly the same thing happens with a different argument. First, Lisp evaluates the argument `(+ 3 4)`, getting 7; then it calls the function `square` with that value. (The function `square` never "sees" the argument as `(+ 3 4)`, it only "sees" the result of evaluating it.) The parameter `length` becomes bound to 7, and the body of the function (`(* length length)`) is evaluated inside of that binding, returning the value 49.

The function `circle` is quite similar. We know the argument (the radius), and we know the formula to use (3 times the square of the radius). The function can be written as:

```
(defun circle (radius)
  (* 3 (* radius radius)))
=> circle
```

Trying it out:

```
(circle 10)
=> 300
(circle (+ 3 4))
=> 147
```

Just as one `let` protects the value we give a variable at the top level, a let `inside` of a `let` will protect the value we give to a variable of the first `let`:

```
(setq number 6)
=> 6
(let (number)            ;first let.
  (setq number 7)        ;number is 7.
  (let (number)          ;second let.
    (setq number 73)     ;number is 73.
    (setq x 8))
                         ;number is 7 again.
  number)
                         ;number is 6 again.
=> 7
number
=> 6
x
=> 8
```

36.5 Writing Simple Functions

For calculating the areas of many squares and circles, it would be useful to have Lisp functions for those things. It would useful to have functions named `square` and `circle` that could be used like this:

```
(square 10)
=> 100
(square (+ 3 4))
=> 49
(circle 10)
=> 300
```

The function `square` takes one argument, the length of a side, and it returns the area of a square with a side of that length. The function `circle` takes one argument, the length of the radius, and returns the area of a circle.

Emacs Lisp does not come with functions `square` and `circle`. But it does let you create them. Creating a function is called *defining* it. The Lisp construct `defun` is used for this purpose. Here's the definition of a new function `square`:

```
(defun square (length)
  (* length length))
```

and the variable will regain its previous value.

Imagine doing these calculations on paper. There is one sheet of paper that the other person's program uses. I have another sheet of paper that I use for doing my own calculations.

When the other person's program runs, it uses its sheet of paper. It might "write" `square = 55` on its paper. When I do my calculations, I put my sheet of paper on top of that one and write `square = 100` on it. When I finish, I'll toss out my sheet of paper. When the other program "looks" at the desk again, it will see the first sheet of paper with `square = 55` on written it—as if I had never changed it.

To make a local binding in Lisp, we use a construct named `let`. We follow it with a list of the variables we want to use locally, then all of the expressions we want to evaluate using those local bindings.

When the `let` is finished being evaluated, it returns the value of the very last expression in it (in the example below: `(+ square circle)`. Then all the local values established by the `let` disappear.

Here's the area calculation done inside of a `let`. (When you evaluate this expression, type the `LFD` only after the final closing parenthsis.)

```
(let (square circle)
   (setq square (* 10 10))
   (setq circle (* 3 square))
   (+ square circle))
=> 400
```

If `square` or `circle` previously had a value, that value remains unchanged afterward:

```
(setq square 1)
=> 1
(let (square circle)
   (setq square (* 10 10))  ;square is now 100
   (setq circle (* 3 square))
   (+ square circle))
=> 400  ;square is 1 once again
square
=> 1
```

We say: "The `let` binds the variable `square` locally."

```
   (+ (+ foo foo) (+ foo foo)))
=> 60

(+ (+ foo foo)
   (+ (+ foo foo)
      (+ foo foo)))
=> 60

(+      (+        foo      foo)   (+ (+ foo
         foo)
   (+ foo      foo)))
=> 60
```

Lisp programmers generally prefer the style used in the second and third expression above. We recommend that you keep all Lisp expressions properly formatted. The `TAB` command in Lisp Mode is defined to indent lines this way. (Look at the binding for the key `TAB`.)

36.4 Binding Variables Locally

Let's say that I wish to find the area of some geometric figures, say the sum of the areas of a square 10 by 10 and a circle with a radius of 10.

I know that the area of a square is the length of one side times itself. The expression for this is `(* 10 10)`. I also know that the area of a circle is pi (which is about 3) times the radius squared. The expression for this is `(* 3 (* 10 10))`. Using Emacs Lisp, I could calculate the desired area as follows:

```
(setq square (* 10 10))
=> 100
(setq circle (* 3 square))
=> 300
(+ square circle)
=> 400
```

If I were the only one to program Emacs, then all would be fine. However, I use programs that other people have written and one of them may use the variable `square`. I'd really rather do my calculations in such a way that I don't have to worry about disturbing any variables.

The way we accomplish this in Lisp is to make what we call a *local binding* for a variable. Having a local binding for a variable means that we can give a variable a value that is only in effect within a certain section of code. When we exit that expression, the "local" value will disappear,

```
=> 50
```

When Lisp looks up the value of a symbol, we say that Lisp is *evaluating* that symbol. When Lisp figures out the value of any expression, we say that Lisp is *evaluating* that expression. `All` expressions evaluate to some value (if they don't cause an error). When we get a value from evaluating an expression, we can also say that the expression *returns* that value, or that the function returns the value.

Thus, to speak proper Lisp, we say: "Lisp evaluated the expression `(+ 5 5)` and returned the value `10`," or, "When given the arguments 5 and 5, the function `+` returns the value 10."

Complex Lisp expressions are made up of simpler expressions. For example, `(* 5 foo)` is a complex expression; Lisp evaluated it and got the value 50. In order to do so, Lisp evaluated each of the arguments to `*` first. Each argument is an expression. Lisp evaluated `5` and got 5. It evaluated `foo` and got 10. Then the function `*` multiplied 5 by 10 to get 50.

If Lisp evaluates the arguments to a function, then we should be able to use any kind of expression as an argument, as long as it evaluates to a suitable value, right? Of course!

In particular, we can use one complex expression as an argument to another complex expression. Below `(* 2 4)` is the second argument to the function `+`:

```
(+ 5 (* 2 4))
=> 13
(* (+ 1 2) (- 10 8))
=> 6
(+ (+ foo foo) (+ (+ foo foo) (+ foo foo)))
=> 60
```

That last example might be a bit difficult to read, just because there are so many parentheses. We can't get rid of parentheses, but we can make them easier to read by adding whitespace. Lisp considers the characters space, newline and tab to be essentially the same: they serve to separate symbols from one another. So we can use them any way we like to make expressions more readable.

Here are some examples which are identical Lisp expressions:

```
(+ (+ foo foo) (+ (+ foo foo) (+ foo foo)))
=> 60

(+ (+ foo foo)
```

Now if we ask Lisp to show us the value of foo:

```
foo
=> 2
```

the value is 2, obeying our previous command.

Just changing the value of a variable and then looking at that value is not terribly interesting. We'd like to be able to do something with those values. *Functions* are constructs in Lisp that do things with values. We tell a function to do its thing by writing a parenthesized expression with the function name following the open-parenthesis.

(Once again, a symbol can be used to name a function, but the symbol is not the function, just as it isn't the value even though it may be used as a variable. A symbol is purely a name. We will often ignore the distinction when it isn't important.)

There are a number of functions in Lisp that are probably quite familiar; for example, the basic functions of arithmetic:

```
(+ 5 5)
=> 10
(- 10 2)
=> 8
(* 5 foo)
=> 10
(/ 10 foo)
=> 5
```

The values that the function operates on are called *arguments*. In the examples above, `5` and `5` are the arguments to the function `+`, and `10` and `2` are the arguments to the function `-`. In the third example, `5` and the value of `foo` are the arguments to `*`.

Notice that when we write `foo` as an argument to `*` (the multiplication function), the actual argument is the `value` of the variable `foo`. This means that if we change the value of `foo`, we'll get a different answer:

```
(setq foo 10)
=> 10
foo
=> 10
(* 5 foo)
```

From now on, after each example of an expression, there will be a line that starts with '=>'. This line shows the value that would result, in that example, from the expression immediately above.

Here are a few more symbols and their values:

```
c-indent-level
=> 2                ;How far to indent each new block of C code
tab-width
=> 8                ;The normal width of a TAB
left-margin
=> 0                ;The location of the left margin
```

When we use a symbol for its value, we refer to the symbol as a *variable*. The values of variables aren't fixed; that's why we call them "variables". `c-indent-level`, `tab-width`, and `left-margin` are all variables and can be changed.

To change the value of `tab-width`, we use the Lisp construct `setq` like this:

```
(setq tab-width 20)
=> 20
```

(terminating with `LFD` of course).

`(setq tab-width 20)` is an expression, so it has a value: 20. It also has the effect of changing the value of `tab-width`.

Now when you insert a tab character into this buffer, it will display as 20 columns wide. Try it! (In Lisp Interaction mode, the `TAB` key is redefined to indent Lisp code properly; to insert an actual tab character, you must type `C-q TAB`.)

We have already seen that an expression may be a symbol or a constant. `(setq tab-width 20)` illustrates another form an expression can take: symbols or constants enclosed in a pair of parentheses.

Let's set the value of some new symbol we think up, say the symbol named `foo`:

```
(setq foo 2)
=> 2
```

36.3 Symbols

Ignoring the nasty details and exceptions, a symbol in Lisp is something that has a name. Normal symbols have names made up of letters, digits, and a few other characters (most often: `-*&^%$@!~_+=/`). Here, each line is a single symbol name:

```
foo1
foobar
foo-bar
+
foo+=_.bar!!!
FOO
lose,win
```

(Upper case characters are different from lower case ones, so `foo` and `FOO` are different symbols.)

Spaces (or tabs or newlines), parentheses and quote marks are among the punctuation characters of Lisp. They are not part of a symbol name.

We will often talk about the names of symbols as if they `were` the symbols. In reality a symbol is not its name any more than I am the three letters '`bil`'. We do talk about me as being "Bil", and we will talk about the symbol named `foo` as if it were `foo`. The distinction can be ignored for the most part.

Symbols can have values associated with them. To demonstrate this, type this into your `*scratch*` buffer

```
fill-column
```

and terminate it with a linefeed (typically a key labeled `LF` or `LFD`). On terminals that don't have an linefeed key, you can always type `C-j`, which is transmitted to the computer as the same character.

Lisp will then print the value associated with the symbol. Typically `fill-column` has the value 70. (`fill-column` is a variable used by the fill commands as the maximum allowable length for a line).

Lisp likes to receive its input in well-defined chunks. We call these chunks *expressions*. When you type `LFD` in the `*scratch*` buffer, Lisp looks backwards to find the previous expression, and then prints the value of that. (Look at the key binding for `LFD`, type `C-h k LFD`.) At the moment, the only kind of expression we know about is a symbol, but we'll see more soon.

36. GNU Emacs Lisp for the Non-Hacker

This section is intended to give people familiar with editing in Emacs enough of an understanding of Lisp to allow them to write their own Emacs commands.

36.1 Lisp History

Lisp (LISt Processing language) was first developed in the late '50s at MIT for use in Artificial Intelligence. Much later, programmers discovered that the great power of the Lisp language made it superior for other purposes as well, such as writing editor 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 60's at MIT's Project MAC. Eventually the implementors of the descendents of Maclisp got together and developed a standard for Lisp systems, called Common Lisp.

GNU Emacs Lisp is largely inspired by Maclisp, and a little by Common Lisp. If you know Common Lisp, you will notice many similarities. But most of the 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.

36.2 Starting with Lisp

Now to Lisp. I will assume that while reading this, you have a terminal at your disposal and that you have called GNU Emacs with no arguments:

```
% emacs
```

When Emacs starts up, the buffer `*scratch*` will be selected. This buffer is in Lisp Interaction mode, a major mode designed especially for you to type simple Lisp programs and execute them.

35. Functions and Variables NOT Documented yet:

This is (should be!!) a list of all functions in the 18.52 that are not documented yet.

35.1 Abbreviations

abbrev-all-caps abbrev-expansion abbrev-mode abbrev-start-location abbrev-start-location-buffer abbrev-symbol abbrev-table-name-list abbrevs abbrevs-changed clear-abbrev-table default-abbrev-mode define-abbrev define-abbrev-table define-global-abbrev define-mode-abbrev expand-abbrev fundamental-mode-abbrev-table global-abbrev-table insert-abbrev-table-description last-abbrev last-abbrev-location last-abbrev-text make-abbrev-table unexpand-abbrev

This function calls the value of terminal-mode-hook if that exists and is non-`nil` after the terminal buffer has been set up and the subprocess started.

Presently with 'termcap' only; if somebody sends us code to make this work with 'terminfo' we will try to use it.

34. Miscellaneous Modes

34.1 Telnet

This should be in the user manual!!

telnet *arg* Command

Open a network login connection to host named HOST (a string). Communication with HOST is recorded in a buffer *HOST-telnet*. Normally input is edited in Emacs and sent a line at a time.

34.2 Terminal Emulator

terminal-emulator *buffer program args* Command

Under a display-terminal emulator in BUFFER, run PROGRAM on arguments ARGS. BUFFER's contents are made an image of the display generated by that program, and any input typed when BUFFER is the current Emacs buffer is sent to that program an keyboard input. Interactively, BUFFER defaults to "*terminal*" and PROGRAM and ARGS are parsed from an input-string using 'sh' (which means that the ~user filename-naming convention doesn't work. Isn't un*x wonderful?)

The emulated display terminal will have height the same as that of the the selected window, and width one less than that window's.

To switch buffers and leave the emulator, or to give commands to the emulator itself (as opposed to the program running under it), type Control-^. The following character is an emulator command. Type Control-^ twice to send it to the subprogram. This escape character may be changed using the variable 'terminal-escape-char'.

Meta characters may not currently be sent through the terminal emulator.

Here is a list of some of the variables which control the behaviour of the emulator – see their documentation for more information: terminal-escape-char, terminal-scrolling, terminal-more-processing, terminal-redisplay-interval.

Before suspending, the symbol `suspend-hook` is examined. If it is bound, and its value is non-`nil`, then the value will be called as a function of no arguments. Emacs won't be suspended unless it returns `nil`.

After Emacs resumes, the symbol `suspend-resume-hook` will be examined. If it is bound and non-`nil`, then the value will be called as a function of no arguments.

`suspend-emacs` returns `nil`.

In the example, notice that `pwd` is not echoed by the unix shell after Emacs is suspended.

```
(suspend-emacs)
=> nil

(setq suspend-hook
    (function (lambda ()
               (not (y-or-n-p "Really suspend? ")))))
=> (lambda nil (not (y-or-n-p "Really suspend? ")))
(setq suspend-resume-hook
    (function (lambda () (message "Resumed!"))))
=> (lambda nil (message "Resumed!"))
(suspend-emacs "pwd")
=> nil
---------- Buffer: Minibuffer ----------
Really suspend? y

---------- Parent Shell ----------
lewis@slug[23] % /user/lewis/manual
lewis@slug[24] % fg

---------- Echo Area ----------
Resumed!
```

send-string-to-terminal *string* Function

This function sends *string* to the terminal without alteration. Control characters in *string* will have terminal-dependent effects. A typical use would be defining function keys on terminals that had down-loadable function key definitions.

The example below might define function key 4 to move forward four characters (`CTL-U CTL-F`).

```
(send-string-to-terminal "\eF4\^U\^F")
=> nil
```

33.5 Getting out of Emacs

kill-emacs &optional *no-query* Function

This function exits the Emacs job and kills it. If *no-query* is supplied and non-`nil`, then Emacs will exit directly. Normally, if there are modified files or subprocesses running, Emacs will ask the user if it should `really` exit.

If Emacs is running noninteractively and *no-query* is an integer, then it returns *no-query* as the unix program's exit code.

```
(kill-emacs)
=> nil

---------- Buffer: Minibuffer ----------
1 modified buffer exists, do you really want to exit? (yes or no) yes
Subprocesses are executing;  kill them and exit? (yes or no) no
---------- Buffer: Minibuffer ----------
```

suspend-emacs *string* Function

This function stops Emacs and returns to the superior process. The Emacs process is still resumable (unlike after calling `kill-emacs`). This function does nothing if Emacs is running under X windows.

If *string* is non-`nil`, its characters are stuffed to be read as terminal input by Emacs's superior shell. The characters in *string* will not be echoed by the superior shell, the results will just appear.

```
=> "Directory /user/lewis/emacs/"
default-directory
=> "/user/lewis/emacs/"
```

33.4 I/O

set-input-mode *interrupt flow* Function

This function sets the mode for reading keyboard input. If