GNU Emacs Lisp Reference Manual

Volume 1
For Emacs Version 24.1
Revision 3.1, May 2012

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

This is edition 3.1 of the GNU Emacs Lisp Reference Manual,
corresponding to Emacs version 24.1.

Copyright © 1990-1996, 1998-2012 Free Software Foundation, Inc.

> Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with the Invariant Sections being "GNU General Public License," with the Front-Cover texts being "A GNU Manual," and with the Back-Cover Texts as in (a) below. A copy of the license is included in the section entitled "GNU Free Documentation License."
>
> (a) The FSF's Back-Cover Text is: "You have the freedom to copy and modify this GNU manual. Buying copies from the FSF supports it in developing GNU and promoting software freedom."

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

Cover art by Etienne Suvasa.

Short Contents

Volume 2

Table of Contents

Volume 2

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 3.1 of the *GNU Emacs Lisp Reference Manual*, corresponding to Emacs version 24.1.

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.

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

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

Please send comments and corrections using `M-x report-emacs-bug`.

1.2 Lisp History

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

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

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

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

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

1.3 Conventions

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

1.3.1 Some Terms

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

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

1.3.2 `nil` and `t`

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

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

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

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

In contexts where a truth value is expected, any non-`nil` value is considered to be *true*. However, `t` is the preferred way to represent the truth value *true*. When you need to choose a value 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 137.

`booleanp` *object* [Function]

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

1.3.3 Evaluation Notation

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

```
(car '(1 2))
     ⇒ 1
```

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

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

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

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 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 '$\boxed{\text{error}}$'. Note that '$\boxed{\text{error}}$' 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 165, 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.

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

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

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

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

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

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

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

1.3.7.2 A Sample Variable Description

A *variable* is a name that can 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 23.1 (i686-pc-linux-gnu, GTK+ Version 2.14.4)
             of 2009-06-01 on cyd.mit.edu"
```

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

`emacs-build-time` [Variable]

The value of this variable indicates the time at which Emacs was built. It is a list of three integers, like the value of `current-time` (see Section 39.5 [Time of Day], page 399, vol. 2).

```
emacs-build-time
     ⇒ (18846 52016 156039)
```

`emacs-version` [Variable]

The value of this variable is the version of Emacs being run. It is a string such as `"23.1.1"`. 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 `"22.0.91.1"`, indicates an unreleased test version.

`emacs-major-version` [Variable]

The major version number of Emacs, as an integer. For Emacs version 23.1, the value is 23.

`emacs-minor-version` [Variable]

The minor version number of Emacs, as an integer. For Emacs version 23.1, the value is 1.

1.5 Acknowledgements

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

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

For a more complete list of contributors, please see the relevant ChangeLog file in the Emacs sources.

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.

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

In most languages, the programmer must declare the data type of each variable, and the type is known by the compiler but not represented in the data. Such type declarations do not exist in Emacs Lisp. A Lisp variable can have any type of value, and it remembers whatever value you store in it, type and all. (Actually, a small number of Emacs Lisp variables can only take on values of a certain type. See Section 11.14 [Variables with Restricted Values], page 162.)

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 274.

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 276, 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 223). It isn't meant for source files, however.

See Section D.7 [Comment Tips], page 453, vol. 2, 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 −536870912 to 536870911 (30 bits; i.e., -2^{29} to $2^{29}-1$) on typical 32-bit machines. (Some machines provide a wider range.) Emacs Lisp arithmetic functions do not check for overflow. Thus `(1+ 536870911)` is −536870912 if Emacs integers are 30 bits.

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

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

As a special exception, if a sequence of digits specifies an integer too large or too small to be a valid integer object, the Lisp reader reads it as a floating-point number (see Section 2.3.2 [Floating Point Type], page 10). For instance, if Emacs integers are 30 bits, `536870912` is read as the floating-point number `536870912.0`.

See Chapter 3 [Numbers], page 33, 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 33, 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 and buffers are currently limited to the range of 0 to 4194303—twenty two bits (see Section 33.4 [Character Codes], page 185, vol. 2). Codes 0 through 127 are ASCII codes; the rest are non-ASCII (see Chapter 33 [Non-ASCII Characters], page 182, vol. 2). 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 456.

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

These sequences which start with backslash are also known as *escape sequences*, because backslash plays the role of an "escape character"; this 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 several types of escape syntax that you can use to specify non-ASCII text characters.

You can specify characters by their Unicode values. `?\u`*`nnnn`* represents a character that maps to the Unicode code point '`U+`*`nnnn`*' (by convention, Unicode code points are given in hexadecimal). There is a slightly different syntax for specifying characters with code points higher than `U+`*`ffff`*: `\U00`*`nnnnnn`* represents the character whose code point is '`U+`*`nnnnnn`*'. The Unicode Standard only defines code points up to '`U+`*`10ffff`*', so if you specify a code point higher than that, 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 only in 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.

```
?\012 ⇒ 10          ?\n ⇒ 10          ?\C-j ⇒ 10
?\101 ⇒ 65          ?A ⇒ 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 `?\xe0` 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 text 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. See Section 21.7.15 [Strings of Events], page 341, for details about `META`-handling in strings.

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

2.3.3.5 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 text terminals do not report the distinction. The Lisp syntax for the shift bit is '`\S-`'; thus, '`?\C-\S-o`' or '`?\C-\S-O`' represents the shifted-control-o character.

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

2.3.4 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. See Section 11.2 [Constant Variables], page 137.

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’.
```

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

As an exception to the rule that a symbol's name serves as its printed representation, ‘##’ is the printed representation for an interned symbol whose name is an empty string. Furthermore, ‘#:*foo*’ is the printed representation for an uninterned symbol whose name is *foo*. (Normally, the Lisp reader interns all symbols; see Section 8.3 [Creating Symbols], page 104.)

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*.

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

Arrays are fixed-length sequences. They are further subdivided into strings, vectors, char-tables and bool-vectors. Vectors can hold elements of any type, whereas string elements must be characters, and bool-vector elements must be `t` or `nil`. Char-tables are like vectors except that they are indexed by any valid character code. The characters in a string can have text properties like characters in a buffer (see Section 32.19 [Text Properties], page 156, vol. 2), but vectors do not support text properties, even when their elements happen to be characters.

Lists, strings and the other array types also share important similarities. For example, all have a length *l*, and all have elements which can be indexed from zero to *l* minus one. Several functions, called sequence functions, accept any kind of sequence. For example, the function `length` reports the length of any kind of sequence. See Chapter 6 [Sequences Arrays Vectors], page 86.

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* any Lisp object. We also say that “the CAR of this cons cell is” whatever object its CAR slot currently holds, and likewise for the CDR.

A *list* is a series of cons cells, linked together so that the CDR slot of each cons cell holds either the next cons cell or the empty list. The empty list is actually the symbol `nil`. See Chapter 5 [Lists], page 64, for details. Because most cons cells are used as part of lists, we refer to any structure made out of cons cells as a *list structure*.

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

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

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

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

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

The names CAR and CDR derive from the history of Lisp. The original Lisp implementation ran on an IBM 704 computer which divided words into two parts, 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)`:

```
    --- ---      --- ---      --- ---
   |   |   |--> |   |   |--> |   |   |--> nil
    --- ---      --- ---      --- ---
     |            |            |
     |            |            |
      --> 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:

```
     ---------------       ----------------       -------------------
    | car   | cdr   |     | car    | cdr   |     | car       | cdr   |
    | rose  |   o-------->| violet |   o-------->| buttercup |  nil  |
    |       |       |     |        |       |     |           |       |
     ---------------       ----------------       -------------------
```

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:

```
     --- ---      --- ---
    |   |   |--> |   |   |--> nil
     --- ---      --- ---
      |            |
      |            |
       --> A        --> nil
```

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:

```
     --- ---      --- ---      --- ---
    |   |   |--> |   |   |--> |   |   |--> nil
     --- ---      --- ---      --- ---
      |            |            |
      |            |            |
      |             --> oak      --> maple
      |
      |     --- ---      --- ---
       --> |   |   |--> |   |   |--> nil
            --- ---      --- ---
             |            |
             |            |
              --> pine     --> needles
```

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

```
     --------------       --------------       --------------
    | car   | cdr  |     | car   | cdr  |     | car   | cdr  |
    |   o   |   o------->| oak   |   o------->| maple |  nil |
    |   |   |      |     |       |      |     |       |      |
     -- | ---------       --------------       --------------
        |
        |
        |        --------------       ----------------
        |       | car   | cdr  |     | car     | cdr  |
         ------>| pine  |   o------->| needles |  nil |
                |       |      |     |         |      |
                 --------------       ----------------
```

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)`:

```
      --- ---
     |   |   |--> violet
      --- ---
       |
       |
        --> rose
```

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:

```
      --- ---      --- ---
     |   |   |--> |   |   |--> buttercup
      --- ---      --- ---
       |            |
       |            |
        --> rose     --> violet
```

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:

```
      --- ---      --- ---
     |   |   |--> |   |   |--> nil
      --- ---      --- ---
       |            |
       |            |
        --> rose     --> violet
```

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 82, 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 48, for functions that operate on strings.

2.3.8.1 Syntax for Strings

The read syntax for a string is a double-quote, an arbitrary number of characters, and another double-quote, `"like this"`. To include a double-quote in a string, precede it with a backslash; thus, `"\""` is a string containing just 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 (see Section 33.1 [Text Representations], page 182, vol. 2). If the string constant is read from a multibyte source, such as a multibyte buffer or string, or a file that would be visited as multibyte, then Emacs reads the non-ASCII character as a multibyte character and automatically makes the string a multibyte string. If the string constant is read from a unibyte source, then Emacs reads the non-ASCII character as unibyte, and makes the string unibyte.

Instead of writing a non-ASCII character literally into a multibyte string, you can write it as its character code using a hex escape, '\x*nnnnnnn*', with as many digits as necessary. (Multibyte non-ASCII character codes are all greater than 256.) You can also specify a character in a multibyte string using the '\u' or '\U' Unicode escape syntax (see Section 2.3.3.2 [General Escape Syntax], page 11). In either case, any character which is not a valid hex digit terminates the 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, '\xe0\ ' represents one character, 'a' with grave accent. '\ ' in a string constant is just like backslash-newline; it does not contribute any character to the string, but it does terminate the preceding hex escape. Using any hex escape in a string (even for an ASCII character) automatically forces the string to be multibyte.

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.

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 2^7 bit of the character in the string. If the string is used in `define-key` or `lookup-key`, this numeric code is translated into the equivalent meta character. See Section 2.3.3 [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 156, vol. 2, 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 90, 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 92, for special functions to operate on char-tables. Uses of char-tables include:

- Case tables (see Section 4.9 [Case Tables], page 61).
- Character category tables (see Section 35.9 [Categories], page 247, vol. 2).
- Display tables (see Section 38.20.2 [Display Tables], page 377, vol. 2).
- Syntax tables (see Chapter 35 [Syntax Tables], page 234, vol. 2).

2.3.11 Bool-Vector Type

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

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

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

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

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

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

2.3.12 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. The printed representation of a hash table specifies its properties and contents, like this:

```
(make-hash-table)
     ⇒ #s(hash-table size 65 test eql rehash-size 1.5
                             rehash-threshold 0.8 data ())
```

See Chapter 7 [Hash Tables], page 97, for more information about hash tables.

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 165).

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 174). A named function in Lisp is just a symbol with a valid function in its function cell (see Section 12.4 [Defining Functions], page 169).

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 170.

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 181, for an explanation of how to write a macro.

Warning: Lisp macros and keyboard macros (see Section 21.16 [Keyboard Macros], page 358) 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

Byte-code function objects are produced by byte-compiling Lisp code (see Chapter 16 [Byte Compilation], page 223). Internally, a byte-code function object is much like a vector;

however, the evaluator handles this data type specially when it appears in a function call. See Section 16.7 [Byte-Code Objects], page 229.

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 213, 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 1, vol. 2). Most buffers hold the contents of a disk file (see Chapter 25 [Files], page 461) 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 18, vol. 2). But a buffer need not be displayed in any window. Each buffer has a designated position called *point* (see Chapter 30 [Positions], page 99, vol. 2); most editing commands act on the contents of the current buffer in the neighborhood of point. At any time, one buffer is the *current buffer*.

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

Many of the standard Emacs functions manipulate or test the characters in the current buffer; a whole chapter in this manual is devoted to describing these functions (see Chapter 32 [Text], page 122, vol. 2).

Several other data structures are associated with each buffer:

- a local syntax table (see Chapter 35 [Syntax Tables], page 234, vol. 2);
- a local keymap (see Chapter 22 [Keymaps], page 360); and,

- a list of buffer-local variable bindings (see Section 11.10 [Buffer-Local Variables], page 150).
- overlays (see Section 38.9 [Overlays], page 315, vol. 2).
- text properties for the text in the buffer (see Section 32.19 [Text Properties], page 156, vol. 2).

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 15, vol. 2.

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 112, vol. 2, 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 18, vol. 2, 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 66, vol. 2, for a description of the functions that work on frames.

2.4.5 Terminal Type

A *terminal* is a device capable of displaying one or more Emacs frames (see Section 2.4.4 [Frame Type], page 25).

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

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

2.4.6 Window Configuration Type

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

Window configurations do not have a read syntax; their print syntax looks like '`#<window-configuration>`'. See Section 28.23 [Window Configurations], page 60, vol. 2, for a description of several functions related to window configurations.

2.4.7 Frame Configuration Type

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

See Section 29.12 [Frame Configurations], page 86, vol. 2, for a description of several functions related to frame configurations.

2.4.8 Process Type

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

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

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

See Chapter 37 [Processes], page 257, vol. 2, for information about functions that create, delete, return information about, send input or signals to, and receive output from processes.

2.4.9 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 284) or output in the echo area (see Section 38.4 [The Echo Area], page 302, vol. 2).

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

See Chapter 19 [Read and Print], page 274, for a description of functions related to streams, including parsing and printing functions.

2.4.10 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 360, for information about creating keymaps, handling prefix keys, local as well as global keymaps, and changing key bindings.

2.4.11 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 315, vol. 2, for information on how you can create and use overlays.

2.4.12 Font Type

A *font* specifies how to display text on a graphical terminal. There are actually three separate font types—*font objects*, *font specs*, and *font entities*—each of which has slightly different properties. None of them have a read syntax; their print syntax looks like '`#<font-object>`', '`#<font-spec>`', and '`#<font-entity>`' respectively. See Section 38.12.12 [Low-Level Font], page 341, vol. 2, for a description of these Lisp objects.

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 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 282.

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 89.
`bool-vector-p`	See Section 6.7 [Bool-Vectors], page 94.
`bufferp`	See Section 27.1 [Buffer Basics], page 1, vol. 2.
`byte-code-function-p`	See Section 2.3.16 [Byte-Code Type], page 22.
`case-table-p`	See Section 4.9 [Case Tables], page 61.
`char-or-string-p`	See Section 4.2 [Predicates for Strings], page 49.
`char-table-p`	See Section 6.6 [Char-Tables], page 92.
`commandp`	See Section 21.3 [Interactive Call], page 321.
`consp`	See Section 5.2 [List-related Predicates], page 64.
`custom-variable-p`	See Section 14.3 [Variable Definitions], page 193.
`display-table-p`	See Section 38.20.2 [Display Tables], page 377, vol. 2.
`floatp`	See Section 3.3 [Predicates on Numbers], page 35.
`fontp`	See Section 38.12.12 [Low-Level Font], page 341, vol. 2.
`frame-configuration-p`	See Section 29.12 [Frame Configurations], page 86, vol. 2.

`frame-live-p`	See Section 29.6 [Deleting Frames], page 82, vol. 2.
`framep`	See Chapter 29 [Frames], page 66, vol. 2.
`functionp`	See Chapter 12 [Functions], page 163.
`hash-table-p`	See Section 7.4 [Other Hash], page 101.
`integer-or-marker-p`	See Section 31.2 [Predicates on Markers], page 113, vol. 2.
`integerp`	See Section 3.3 [Predicates on Numbers], page 35.
`keymapp`	See Section 22.4 [Creating Keymaps], page 363.
`keywordp`	See Section 11.2 [Constant Variables], page 137.
`listp`	See Section 5.2 [List-related Predicates], page 64.
`markerp`	See Section 31.2 [Predicates on Markers], page 113, vol. 2.
`wholenump`	See Section 3.3 [Predicates on Numbers], page 35.
`nlistp`	See Section 5.2 [List-related Predicates], page 64.
`numberp`	See Section 3.3 [Predicates on Numbers], page 35.
`number-or-marker-p`	See Section 31.2 [Predicates on Markers], page 113, vol. 2.
`overlayp`	See Section 38.9 [Overlays], page 315, vol. 2.
`processp`	See Chapter 37 [Processes], page 257, vol. 2.
`sequencep`	See Section 6.1 [Sequence Functions], page 86.
`stringp`	See Section 4.2 [Predicates for Strings], page 49.
`subrp`	See Section 12.8 [Function Cells], page 175.
`symbolp`	See Chapter 8 [Symbols], page 102.
`syntax-table-p`	See Chapter 35 [Syntax Tables], page 234, vol. 2.
`vectorp`	See Section 6.4 [Vectors], page 90.
`window-configuration-p`	See Section 28.23 [Window Configurations], page 60, vol. 2.
`window-live-p`	See Section 28.6 [Deleting Windows], page 31, vol. 2.
`windowp`	See Section 28.1 [Basic Windows], page 18, vol. 2.
`booleanp`	See Section 1.3.2 [nil and t], page 2.

`string-or-null-p`
See Section 4.2 [Predicates for Strings], page 49.

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

`type-of` *object* [Function]

This function returns a symbol naming the primitive type of *object*. The value is one of the symbols `bool-vector`, `buffer`, `char-table`, `compiled-function`, `cons`, `float`, `font-entity`, `font-object`, `font-spec`, `frame`, `hash-table`, `integer`, `marker`, `overlay`, `process`, `string`, `subr`, `symbol`, `vector`, `window`, 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 functions that test for equality between two objects. Other functions test equality of contents between objects of specific types, e.g. strings. For these predicates, see the appropriate chapter describing the data type.

`eq` *object1 object2* [Function]

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

If *object1* and *object2* are integers with the same value, they are considered to be the same object (i.e. `eq` returns `t`). If *object1* and *object2* are symbols with the same name, they are normally the same object—but see Section 8.3 [Creating Symbols], page 104 for exceptions. 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 "" "")
     ⇒ t
;; This exception occurs because Emacs Lisp
;; makes just one multibyte empty string, to save space.

(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, and `nil` otherwise. Whereas `eq` tests if its arguments are the same object, `equal` looks inside nonidentical arguments to see if their elements or contents are the same. So, if two objects are `eq`, they are `equal`, but the converse is not always true.

```
(equal 'foo 'foo)
     ⇒ t

(equal 456 456)
     ⇒ t

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

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

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

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

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

Comparison of strings is case-sensitive, but does not take account of text properties—it compares only the characters in the strings. See Section 32.19 [Text Properties], page 156, vol. 2. Use `equal-including-properties` to also compare text properties. For technical reasons, a unibyte string and a multibyte string are `equal` if and only if they contain the same sequence of character codes and all these codes are either in the range 0 through 127 (ASCII) or 160 through 255 (`eight-bit-graphic`). (see Section 33.1 [Text Representations], page 182, vol. 2).

```
(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).

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

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

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

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: 1.5e2 equals 150; in this example, '`e2`' 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 −536870912 to 536870911 (30 bits; i.e., -2^{29} to $2^{29} - 1$), but some machines provide a wider range. Many examples in this chapter assume that an integer has 30 bits and that floating point numbers are IEEE double precision.

The Lisp reader reads an integer as a sequence of digits with optional initial sign and optional final period. An integer that is out of the Emacs range is treated as a floating-point number.

```
 1                  ; The integer 1.
 1.                 ; The integer 1.
+1                  ; Also the integer 1.
-1                  ; The integer −1.
 1073741825         ; The floating point number 1073741825.0.
 0                  ; The integer 0.
-0                  ; 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 '*`radix`*`r`' to specify radix *radix*. Case is not significant for the letter that specifies the radix. Thus, '`#b`*`integer`*' reads *integer* in binary, and '`#`*`radix`*`r`*`integer`*' reads *integer* in radix *radix*. Allowed values of *radix* run from 2 to 36. 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 42), it is often helpful to view the numbers in their binary form.

In 30-bit binary, the decimal integer 5 looks like this:

```
0000...000101 (30 bits total)
```

(The '`...`' stands for enough bits to fill out a 30-bit word; in this case, '`...`' stands for twenty 0 bits. Later examples also use the '`...`' notation to make binary integers easier to read.)

The integer −1 looks like this:

```
1111...111111 (30 bits total)
```

−1 is represented as 30 ones. (This is called *two's complement* notation.)

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

```
1111...111011 (30 bits total)
```

In this implementation, the largest 30-bit binary integer value is 536,870,911 in decimal. In binary, it looks like this:

```
0111...111111 (30 bits total)
```

Since the arithmetic functions do not check whether integers go outside their range, when you add 1 to 536,870,911, the value is the negative integer $-536,870,912$:

```
(+ 1 536870911)
     ⇒ -536870912
     ⇒ 1000...000000 (30 bits total)
```

Many of the functions described in this chapter accept markers for arguments in place of numbers. (See Chapter 31 [Markers], page 112, vol. 2.) 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.

See Section 33.4 [Character Codes], page 185, vol. 2, for the maximum value of a valid character codepoint.

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. Emacs uses the IEEE floating point standard where possible (the standard is supported by most modern computers).

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`’.

Emacs Lisp treats `-0.0` as equal to ordinary zero (with respect to `equal` and `=`), even though the two are distinguishable in the IEEE floating point standard.

The IEEE floating point standard supports positive infinity and negative infinity as floating point values. It also provides for a class of values called NaN or “not-a-number”; numerical functions return such values in cases where there is no correct answer. For example, `(/ 0.0 0.0)` returns a NaN. (NaN values can also carry a sign, but for practical purposes there’s no significant difference between different NaN values in Emacs Lisp.) 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`’.

`isnan` *number* [Function]

This predicate tests whether its argument is NaN, and returns `t` if so, `nil` otherwise. The argument must be a number.

The following functions are specialized for handling floating point numbers:

`frexp` *x* [Function]

This function returns a cons cell `(`*sig* `.` *exp*`)`, where *sig* and *exp* are respectively the significand and exponent of the floating point number *x*:

```
x = sig * 2^exp
```

sig is a floating point number between 0.5 (inclusive) and 1.0 (exclusive). If *x* is zero, the return value is `(0 . 0)`.

`ldexp` *sig* **&optional** *exp* [Function]

This function returns a floating point number corresponding to the significand *sig* and exponent *exp*.

`copysign` *x1 x2* [Function]

This function copies the sign of *x2* to the value of *x1*, and returns the result. *x1* and *x2* must be floating point numbers.

`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 113, vol. 2.

`floatp` *object* [Function]

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

`integerp` *object* [Function]

This predicate tests whether its argument is an integer, and returns `t` if so, `nil` otherwise.

`numberp` *object* [Function]

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

`natnump` *object* [Function]

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

This is a synonym for `natnump`.

`zerop` *number* [Function]

This predicate tests whether its argument is zero, and returns `t` if so, `nil` otherwise. The argument must be a number.

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

3.4 Comparison of Numbers

To test numbers for numerical equality, you should normally use `=`, 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.

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

`/=` *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+ 536870911)` may evaluate to −536870912, 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 128.) 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 38.

3.7 Rounding Operations

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

`ffloor` *float* [Function]

This function rounds *float* to the next lower integral value, and returns that value as a floating point number.

`fceiling` *float* [Function]

This function rounds *float* to the next higher integral value, and returns that value as a floating point number.

`ftruncate` *float* [Function]

This function rounds *float* towards zero to an integral value, and returns that value as a floating point number.

`fround` *float* [Function]

This function rounds *float* to the nearest integral value, and returns that value as a floating point number.

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 536,870,911 produces −2 in the 30-bit implementation:

```
(lsh 536870911 1)          ; left shift
     ⇒ -2
```

In binary, the argument looks like this:

```
;; Decimal 536,870,911
0111...111111 (30 bits total)
```

which becomes the following when left shifted:

```
;; Decimal −2
1111...111110 (30 bits total)
```

`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.
1111...111010 (30 bits total)
     ⇒
1111...111101 (30 bits total)
```

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

```
(lsh -6 -1) ⇒ 536870909
;; Decimal −6 becomes decimal 536,870,909.
1111...111010 (30 bits total)
     ⇒
0111...111101 (30 bits total)
```

Here are other examples:

```
                    ;        30-bit binary values

(lsh 5 2)           ;   5  =  0000...000101
     ⇒ 20           ;      =  0000...010100
(ash 5 2)
     ⇒ 20
(lsh -5 2)          ;  -5  =  1111...111011
     ⇒ -20          ;      =  1111...101100
(ash -5 2)
     ⇒ -20
(lsh 5 -2)          ;   5  =  0000...000101
     ⇒ 1            ;      =  0000...000001
(ash 5 -2)
     ⇒ 1
(lsh -5 -2)         ;  -5  =  1111...111011
     ⇒ 268435454
                    ;      =  0011...111110
(ash -5 -2)         ;  -5  =  1111...111011
     ⇒ -2           ;      =  1111...111110
```

`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.

```
                         ;      30-bit binary values

(logand 14 13)           ; 14  =  0000...001110
                         ; 13  =  0000...001101
     ⇒ 12                ; 12  =  0000...001100

(logand 14 13 4)         ; 14  =  0000...001110
                         ; 13  =  0000...001101
                         ;  4  =  0000...000100
     ⇒ 4                 ;  4  =  0000...000100

(logand)
     ⇒ -1                ; -1  =  1111...111111
```

`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.

```
                         ;      30-bit binary values

(logior 12 5)            ; 12  =  0000...001100
                         ;  5  =  0000...000101
     ⇒ 13                ; 13  =  0000...001101

(logior 12 5 7)          ; 12  =  0000...001100
                         ;  5  =  0000...000101
                         ;  7  =  0000...000111
     ⇒ 15                ; 15  =  0000...001111
```

`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.

```
                         ;      30-bit binary values

(logxor 12 5)            ; 12  =  0000...001100
                         ;  5  =  0000...000101
     ⇒ 9                 ;  9  =  0000...001001

(logxor 12 5 7)          ; 12  =  0000...001100
                         ;  5  =  0000...000101
                         ;  7  =  0000...000111
     ⇒ 14                ; 14  =  0000...001110
```

`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  =  0000...000101 (30 bits total)
;; becomes
;; -6  =  1111...111010 (30 bits total)
```

3.9 Standard Mathematical Functions

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

`sin` *arg* [Function]
`cos` *arg* [Function]
`tan` *arg* [Function]

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

`asin` *arg* [Function]

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

`acos` *arg* [Function]

The value of `(acos arg)` is a number between 0 and π (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*.

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

This function returns the logarithm of *arg*, with base *base*. If you don't specify *base*, the natural base e is used. If *arg* 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)` ≡ `(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.

In addition, Emacs defines the following common mathematical constants:

`float-e` [Variable]

The mathematical constant e (2.71828...).

`float-pi` [Variable]

The mathematical constant *pi* (3.14159. . .).

3.10 Random Numbers

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

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 21.7.15 [Strings of Events], page 341, 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. See Section 33.4 [Character Codes], page 185, vol. 2, for details about character representation in Emacs.

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 86.) 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 89). However, note that `length` should *not* be used for computing the width of a string on display; use `string-width` (see Section 38.10 [Width], page 323, vol. 2) instead.

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 182, vol. 2). For most Lisp programming, you don't need to be concerned with these two representations.

Sometimes key sequences are represented as unibyte strings. When a unibyte string is a key sequence, string elements in the range 128 to 255 represent meta characters (which are large integers) rather than character codes in the range 128 to 255. Strings cannot hold characters that have the hyper, super or alt modifiers; they can hold ASCII control characters, but no other control characters. They do not distinguish case in ASCII control characters. If you want to store such characters in a sequence, such as a key sequence, you must use a vector instead of a string. See Section 2.3.3 [Character Type], page 10, for more information about keyboard input characters.

Strings are useful for holding regular expressions. You can also match regular expressions against strings with `string-match` (see Section 34.4 [Regexp Search], page 221, vol. 2). The functions `match-string` (see Section 34.6.2 [Simple Match Data], page 226, vol. 2) and `replace-match` (see Section 34.6.1 [Replacing Match], page 225, vol. 2) 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 156, vol. 2. 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 122, vol. 2, 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 182, vol. 2, 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 86, and Section 6.2 [Arrays], page 88.

`stringp` *object* [Function]

This function returns `t` if *object* is a string, `nil` otherwise.

`string-or-null-p` *object* [Function]

This function returns `t` if *object* is a string or `nil`. It returns `nil` otherwise.

`char-or-string-p` *object* [Function]

This function returns `t` if *object* is a string or a character (i.e., an integer), `nil` otherwise.

4.3 Creating Strings

The following functions create strings, either from scratch, or by putting strings together, or by taking them apart.

`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 `make-vector` (see Section 6.4 [Vectors], page 90) and `make-list` (see Section 5.4 [Building Lists], page 68).

`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"
```

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

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

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

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

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

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

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

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

But we recommend `copy-sequence` for this purpose (see Section 6.1 [Sequence Functions], page 86).

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 156, vol. 2.

`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 123, vol. 2), 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)
          ⇒ ""
```

This function always constructs a new string that is not `eq` to any existing string, except when the result is the empty string (to save space, Emacs makes only one empty multibyte string).

For information about other concatenation functions, see the description of `mapconcat` in Section 12.6 [Mapping Functions], page 172, `vconcat` in Section 6.5 [Vector Functions], page 91, and `append` in Section 5.4 [Building Lists], page 68. For concatenating individual command-line arguments into a string to be used as a shell command, see Section 37.2 [Shell Arguments], page 258, vol. 2.

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

This function splits *string* into substrings based on the regular expression *separators* (see Section 34.3 [Regular Expressions], page 211, vol. 2). Each match for *separators* defines a splitting point; the substrings between splitting points are made into a list, which is returned.

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

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

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

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

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

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

If you need to split a string into a list of individual command-line arguments suitable for `call-process` or `start-process`, see Section 37.2 [Shell Arguments], page 258, vol. 2.

`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 89). `(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 the symbol names are used. Case is always significant, regardless of `case-fold-search`.

This function is equivalent to `equal` for comparing two strings (see Section 2.7 [Equality Predicates], page 30). In particular, the text properties of the two strings are ignored. But if either argument is not a string or symbol, an error is signaled.

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

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 is converted to a multibyte string, all characters with codes in the range 160 through 255 are 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 182, vol. 2.

`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 182, vol. 2).

```
(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<`.

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

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

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

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 182, vol. 2) 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 82.

See also the function `compare-buffer-substrings` in Section 32.3 [Comparing Text], page 125, vol. 2, 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 221, vol. 2.

4.6 Conversion of Characters and Strings

This section describes functions for converting between characters, strings and integers. `format` (see Section 4.7 [Formatting Strings], page 57) and `prin1-to-string` (see Section 19.5 [Output Functions], page 279) can also convert Lisp objects into strings. `read-from-string` (see Section 19.3 [Input Functions], page 276) can “convert” a string representation of a Lisp object into an object. The functions `string-to-multibyte` and `string-to-unibyte` convert the text representation of a string (see Section 33.2 [Converting Representations], page 183, vol. 2).

See Chapter 24 [Documentation], page 451, for functions that produce textual descriptions of text characters and general input events (`single-key-description` and `text-char-description`). These are used primarily for making help messages.

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

This function returns a string consisting of the printed base-ten representation of *number*, 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 57.

`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.

`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 49.

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

This function returns the first character in *string*. This mostly identical to `(aref string 0)`, except that it returns 0 if the string is empty. (The value is also 0 when the first character of *string* is the null character, ASCII code 0.) This function may be eliminated in the future if it does not seem useful enough to retain.

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

`concat` This function converts a vector or a list into a string. See Section 4.3 [Creating Strings], page 49.

`vconcat` This function converts a string into a vector. See Section 6.5 [Vector Functions], page 91.

`append` This function converts a string into a list. See Section 5.4 [Building Lists], page 68.

`byte-to-string`
This function converts a byte of character data into a unibyte string. See Section 33.2 [Converting Representations], page 183, vol. 2.

4.7 Formatting Strings

Formatting means constructing a string by substituting computed values at various places in a constant string. This constant string controls how the other values are printed, as well as where they appear; it is called a *format string*.

Formatting is often useful for computing messages to be displayed. In fact, the functions `message` and `error` provide the same formatting feature described here; they differ from `format` 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 `princ`, not `prin1`—see Section 19.5 [Output Functions], page 279). Thus, strings are represented by their contents alone, with no '`"`' characters, and symbols appear without '`\`' characters.

If the object is a string, its text properties are copied into the output. The text properties of the '`%s`' itself are also copied, but those of the object take priority.

'`%S`' Replace the specification with the printed representation of the object, made with quoting (that is, using `prin1`—see Section 19.5 [Output Functions], page 279). 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 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 width specifier is ignored for the '%%' specification. Any padding introduced by the width specifier normally consists of spaces inserted on the left:

```
(format "%5d is padded on the left with spaces" 123)
     ⇒ "  123 is padded on the left with spaces"
```

If the width is too small, `format` does not truncate the object's printed representation. Thus, you can use a width to specify a minimum spacing between columns with no risk of losing information. In the following three examples, '%7s' specifies a minimum width of 7. In the first case, the string inserted in place of '%7s' has only 3 letters, and needs 4 blank spaces as padding. In the second case, the string `"specification"` is 13 letters wide but is not truncated.

```
(format "The word '%7s' has %d letters in it."
        "foo" (length "foo"))
     ⇒ "The word '    foo' has 3 letters in it."
(format "The word '%7s' has %d letters in it."
        "specification" (length "specification"))
     ⇒ "The word 'specification' has 13 letters in it."
```

Immediately after the '%' and before the optional width specifier, you can also put certain *flag characters*.

The 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.) These flags are useful for ensuring that positive numbers and negative numbers use the same number of columns. They are ignored except for '%d', '%e', '%f', '%g', and if both flags are used, '+' takes precedence.

The flag '#' specifies an "alternate form" which depends on the format in use. For '%o', it ensures that the result begins with a '0'. For '%x' and '%X', it prefixes the result with '0x' or '0X'. For '%e', '%f', and '%g', the '#' flag means include a decimal point even if the precision is zero.

The flag '0' ensures that the padding consists of '0' characters instead of spaces. This flag is ignored for non-numerical specification characters like '%s', '%S' and '%c'. These specification characters accept the '0' flag, but still pad with *spaces*.

The flag '-' causes the padding inserted by the width specifier, if any, to be inserted on the right rather than the left. If both '-' and '0' are present, the '0' flag is ignored.

```
(format "%06d is padded on the left with zeros" 123)
     ⇒ "000123 is padded on the left with zeros"

(format "%-6d is padded on the right" 123)
     ⇒ "123    is padded on the right"

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

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.

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 61).

These functions do not modify the strings that are passed to them as arguments.

The examples below use the characters '`X`' and '`x`' which have ASCII codes 88 and 120 respectively.

`downcase` *string-or-char* [Function]

This function converts *string-or-char*, which should be either a character or a string, to lower case.

When *string-or-char* is a string, this function returns a new string in which each letter in the argument that is upper case is converted to lower case. When *string-or-char* is a character, this function returns the corresponding lower case character (an integer); if the original character is lower case, or is not a letter, the return value is equal to the original character.

```
(downcase "The cat in the hat")
     ⇒ "the cat in the hat"

(downcase ?X)
     ⇒ 120
```

`upcase` *string-or-char* [Function]

This function converts *string-or-char*, which should be either a character or a string, to upper case.

When *string-or-char* is a string, this function returns a new string in which each letter in the argument that is lower case is converted to upper case. When *string-or-char* is a character, this function returns the corresponding upper case character (an integer); if the original character is upper case, or is not a letter, the return value is equal to the original character.

```
(upcase "The cat in the hat")
     ⇒ "THE CAT IN THE HAT"

(upcase ?x)
     ⇒ 88
```

`capitalize` *string-or-char* [Function]

This function capitalizes strings or characters. If *string-or-char* is a string, the function returns a new string whose contents are a copy of *string-or-char* in which each word has been capitalized. This means that the first character of each word is converted to upper case, and the rest are converted to lower case.

The definition of a word is any sequence of consecutive characters that are assigned to the word constituent syntax class in the current syntax table (see Section 35.2.1 [Syntax Class Table], page 235, vol. 2).

When *string-or-char* is a character, this function does the same thing as `upcase`.

```
(capitalize "The cat in the hat")
     ⇒ "The Cat In The Hat"

(capitalize "THE 77TH-HATTED CAT")
     ⇒ "The 77th-Hatted Cat"
```

```
(capitalize ?x)
     ⇒ 88
```

`upcase-initials` *string-or-char* [Function]

If *string-or-char* is a string, this function capitalizes the initials of the words in *string-or-char*, without altering any letters other than the initials. It returns a new string whose contents are a copy of *string-or-char*, in which each word has had its initial letter converted to upper case.

The definition of a word is any sequence of consecutive characters that are assigned to the word constituent syntax class in the current syntax table (see Section 35.2.1 [Syntax Class Table], page 235, vol. 2).

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 53, 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 155, vol. 2). 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 92) whose subtype is `case-table`. This char-table maps each character into the corresponding lower case character. It has three extra slots, which hold related tables:

upcase The upcase table maps each character into the corresponding upper case character.

canonicalize
The canonicalize table maps all of a set of case-related characters into a particular member of that set.

equivalences
The equivalences table maps each one of a set of case-related characters into the next character in that set.

In simple cases, all you need to specify is the mapping to lower-case; the three related tables will be calculated automatically from that one.

For some languages, upper and lower case letters are not in one-to-one correspondence. There may be two different lower case letters with the same upper case equivalent. In these cases, you need to specify the maps for both lower case and upper case.

The extra table *canonicalize* maps each character to a canonical equivalent; any two characters that are related by case-conversion have the same canonical equivalent character. For example, since ‘`a`’ and ‘`A`’ are related by case-conversion, they should have the same canonical equivalent character (which should be either '`a`' for both of them, or '`A`' for both of them).

The extra table *equivalences* is a map that cyclically permutes each equivalence class (of characters with the same canonical equivalent). (For ordinary ASCII, this would map '`a`' into '`A`' and '`A`' into '`a`', and likewise for each set of equivalent characters.)

When constructing a case table, you can provide `nil` for *canonicalize*; then Emacs fills in this slot from the lower case and upper case mappings. You can also provide `nil` for *equivalences*; then Emacs fills in this slot from *canonicalize*. In a case table that is actually in use, those components are non-`nil`. Do not try to specify *equivalences* without also specifying *canonicalize*.

Here are the functions for working with case tables:

`case-table-p` *object* [Function]

This predicate returns non-`nil` if *object* is a valid case table.

`set-standard-case-table` *table* [Function]

This function makes *table* the standard case table, so that it will be used in any buffers created subsequently.

`standard-case-table` [Function]

This returns the standard case table.

`current-case-table` [Function]

This function returns the current buffer's case table.

`set-case-table` *table* [Function]

This sets the current buffer's case table to *table*.

`with-case-table` *table body...* [Macro]

The `with-case-table` macro saves the current case table, makes *table* the current case table, evaluates the *body* forms, and finally restores the case table. The return value is the value of the last form in *body*. The case table is restored even in case of an abnormal exit via `throw` or error (see Section 10.5 [Nonlocal Exits], page 126).

Some language environments 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 234, vol. 2. Normally you would use these functions to change the standard case table.

`set-case-syntax-pair` *uc lc case-table* [Function]

This function specifies a pair of corresponding letters, one upper case and one lower case.

`set-case-syntax-delims` *l r case-table* [Function]

This function makes characters *l* and *r* a matching pair of case-invariant delimiters.

`set-case-syntax` *char syntax case-table* [Function]

This function makes *char* case-invariant, with syntax *syntax*.

`describe-buffer-case-table` [Command]

This command displays a description of the contents of the current buffer's case table.

5 Lists

A *list* represents a sequence of zero or more elements (which may be any Lisp objects). The important difference between lists and vectors is that two or more lists can share part of their structure; in addition, you can insert or delete elements in a list without copying the whole list.

5.1 Lists and Cons Cells

Lists in Lisp are not a primitive data type; they are built up from *cons cells* (see Section 2.3.6 [Cons Cell Type], page 14). 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 (this asymmetry between CAR and CDR is entirely a matter of convention; at the level of cons cells, the CAR and CDR slots have similar properties). Hence, the CDR slot of each cons cell in a list refers to the following cons cell.

Also by convention, the CDR of the last cons cell in a list is `nil`. We call such a `nil`-terminated structure a *true list*. In Emacs Lisp, the symbol `nil` is both a symbol and a list with no elements. For convenience, the symbol `nil` is considered to have `nil` as its CDR (and also as its CAR).

Hence, the CDR of a true list is always a true list. The CDR of a nonempty true list is a true list containing all the elements except the first.

If the CDR of a list’s last cons cell is some value other than `nil`, we call the structure a *dotted list*, since its printed representation would use dotted pair notation (see Section 2.3.6.2 [Dotted Pair Notation], page 17). 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 a program doesn’t look far enough down the list to see the CDR of the final cons cell, it won’t care. However, some functions that operate on lists demand 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, we refer to any structure made out of cons cells as a *list structure*.

5.2 Predicates on Lists

The following predicates test whether a Lisp object is an atom, whether it is a cons cell or is a list, or whether it is the distinguished object `nil`. (Many of these predicates can be defined in terms of the others, but they are used so often that it is worth having them.)

`consp` *object* [Function]

This function returns `t` if *object* is a cons cell, `nil` otherwise. `nil` is not a cons cell, although it *is* a list.

`atom` *object* [Function]

This function returns `t` if *object* is an atom, `nil` otherwise. All objects except cons cells are atoms. The symbol `nil` is an atom and is also a list; it is the only Lisp object that is both.

```
(atom object) ≡ (not (consp object))
```

`listp` *object* [Function]

This function returns `t` if *object* is a cons cell or `nil`. Otherwise, it returns `nil`.

```
(listp '(1))
     ⇒ t
(listp '())
     ⇒ t
```

`nlistp` *object* [Function]

This function is the opposite of `listp`: it returns `t` if *object* is not a list. Otherwise, it returns `nil`.

```
(listp object) ≡ (not (nlistp object))
```

`null` *object* [Function]

This function returns `t` if *object* is `nil`, and returns `nil` otherwise. This function is identical to `not`, but as a matter of clarity we use `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 123).

```
(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*. In other words, it returns the CAR of *cons-cell*.

As a special case, if *cons-cell* is `nil`, this function returns `nil`. Therefore, any list is a valid argument. An error is signaled if the argument is not a cons cell or `nil`.

```
(car '(a b c))
     ⇒ a
(car '())
     ⇒ nil
```

`cdr` *cons-cell* [Function]

This function returns the value referred to by the second slot of the cons cell *cons-cell*. In other words, it returns the CDR of *cons-cell*.

As a special case, if *cons-cell* is `nil`, this function returns `nil`; therefore, any list is a valid argument. An error is signaled if the argument is not a cons cell or `nil`.

```
(cdr '(a b c))
     ⇒ (b c)
(cdr '())
     ⇒ nil
```

`car-safe` *object* [Function]

This function lets you take the CAR of a cons cell while avoiding errors for other data types. It returns the CAR of *object* if *object* is a cons cell, `nil` otherwise. This is in contrast to `car`, which signals an error if *object* is not a list.

```
(car-safe object)
≡
(let ((x object))
  (if (consp x)
      (car x)
    nil))
```

`cdr-safe` *object* [Function]

This function lets you take the CDR of a cons cell while avoiding errors for other data types. It returns the CDR of *object* if *object* is a cons cell, `nil` otherwise. This is in contrast to `cdr`, which signals an error if *object* is not a list.

```
(cdr-safe object)
≡
(let ((x object))
  (if (consp x)
      (cdr x)
    nil))
```

`pop` *listname* [Macro]

This macro 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)
```

For the `pop` macro, which removes an element from a list, See Section 5.5 [List Variables], page 71.

`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 86.

`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 86.

`caar` *cons-cell* [Function]

This is the same as `(car (car cons-cell))`.

`cadr` *cons-cell* [Function]

This is the same as `(car (cdr cons-cell))` or `(nth 1 cons-cell)`.

`cdar` *cons-cell* [Function]

This is the same as `(cdr (car cons-cell))`.

`cddr` *cons-cell* [Function]

This is the same as `(cdr (cdr` *`cons-cell`*`))` or `(nthcdr 2` *`cons-cell`*`)`.

`butlast` *x* **&optional** *n* [Function]

This function returns the list *x* with the last element, or the last *n* elements, removed. If *n* is greater than zero it makes a copy of the list so as not to damage the original list. In general, `(append (butlast` *`x n`*`) (last` *`x n`*`))` will return a list equal to *x*.

`nbutlast` *x* **&optional** *n* [Function]

This is a version of `butlast` that works by destructively modifying the `cdr` of the appropriate element, rather than making a copy of the list.

5.4 Building Cons Cells and Lists

Many functions build lists, as lists reside at the very heart of Lisp. `cons` is the fundamental list-building function; however, it is interesting to note that `list` is used more times in the source code for Emacs than `cons`.

`cons` *object1 object2* [Function]

This function is the most basic function for building new list structure. It creates a new cons cell, making *object1* the CAR, and *object2* the CDR. It then returns the new cons cell. The arguments *object1* and *object2* may be any Lisp objects, but most often *object2* is a list.

```
(cons 1 '(2))
     ⇒ (1 2)
(cons 1 '())
     ⇒ (1)
(cons 1 2)
     ⇒ (1 . 2)
```

`cons` is often used to add a single element to the front of a list. This is called *consing the element onto the list.*[1] For example:

```
(setq list (cons newelt list))
```

Note that there is no conflict between the variable named `list` used in this example and the function named `list` described below; any symbol can serve both purposes.

`list` **&rest** *objects* [Function]

This function creates a list with *objects* as its elements. The resulting list is always `nil`-terminated. If no *objects* are given, the empty list is returned.

```
(list 1 2 3 4 5)
     ⇒ (1 2 3 4 5)
(list 1 2 '(3 4 5) 'foo)
     ⇒ (1 2 (3 4 5) foo)
(list)
     ⇒ nil
```

[1] There is no strictly equivalent way to add an element to the end of a list. You can use `(append` *`listname`* `(list` *`newelt`*`))`, which creates a whole new list by copying *listname* and adding *newelt* to its end. Or you can use `(nconc` *`listname`* `(list` *`newelt`*`))`, which modifies *listname* by following all the CDRs and then replacing the terminating `nil`. Compare this to adding an element to the beginning of a list with `cons`, which neither copies nor modifies the list.

`make-list` *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 49).

```
(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 76, 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.

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:

```
more-trees                trees
|                           |
|     --- ---      --- ---   -> --- ---      --- ---
 --> |   |   |--> |   |   |--> |   |   |--> |   |   |--> nil
      --- ---      --- ---      --- ---      --- ---
       |            |            |            |
       |            |            |            |
        --> maple    -->birch     --> pine     --> oak
```

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 86.

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 170), 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)
```

For the `pop` macro, which removes the first element from a list, See Section 5.3 [List Elements], page 65.

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.
     ⇒ (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 necessary.

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                           ; Both lists are changed.
     ⇒ (a foo c)
x2
     ⇒ (z foo c)

;; Replace the CAR of a link that is not shared.
(setcar x1 'baz)
     ⇒ baz
x1                           ; Only one list is changed.
     ⇒ (baz foo c)
x2
     ⇒ (z foo c)
```

Here is a graphical depiction of the shared structure of the two lists in the variables `x1` and `x2`, showing why replacing `b` changes them both:

```
        --- ---        --- ---      --- ---
x1---> |   |   |----> |   |   |--> |   |   |--> nil
        --- ---        --- ---      --- ---
         |        -->   |            |
         |       |      |            |
          --> a  |       --> b        --> c
                 |
       --- ---   |
x2--> |   |   |--
       --- ---
        |
        |
         --> z
```

Here is an alternative form of box diagram, showing the same relationship:

```
x1:
 --------------       --------------       --------------
| car   | cdr  |     | car   | cdr  |     | car   | cdr  |
|   a   |   o------->|   b   |   o------->|   c   |  nil |
|       |      |  -->|       |      |     |       |      |
 --------------   |   --------------       --------------
                  |
x2:               |
 --------------   |
| car   | cdr  |  |
|   z   |   o----
|       |      |
 --------------
```

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:

```
                   --------------------
                  |                    |
 --------------   |    --------------  |    --------------
| car   | cdr  |  |   | car   | cdr  |  -->| car   | cdr  |
|   a   |   o-----    |   b   |   o-------->|   c   |  nil |
|       |      |      |       |      |     |       |      |
 --------------        --------------       --------------
```

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:

```
 ------------      ------------     -----------
| car | cdr  |     |car | cdr |    | car | cdr |
|  a  |  o   |  -->|  b |   o---->|  c  | nil |
|     |  |   | |   |    |     |    |     |     |
 --------| --  |   ------------     -----------
         |     |
    -----       -------
   |                   |
   |   --------------  |
   |   | car | cdr  |  |
    -->|  d  |   o------
       |     |      |
       --------------
```

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 68), 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:

```
Original list head:                       Reversed list:
 -------------        -------------        ------------
| car  | cdr  |      | car  | cdr  |      | car | cdr  |
|   a  |  nil |<--   |   b  |   o  |<--   |   c |   o  |
|      |      |   |  |      |   |  |   |  |     |   |  |
 -------------    |   --------- | -    |   -------- | -
                  |             |      |            |
                   -------------        ------------
```

`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 146, vol. 2, for more functions that perform sorting. See `documentation` in Section 24.2 [Accessing Documentation], page 452, 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. Although standard GNU

Emacs Lisp does not have them, the ‘`cl`’ library provides versions. See Section “Overview” in *Common Lisp Extensions*.

`memq` *object list* [Function]

This function tests to see whether *object* is a member of *list*. If it is, `memq` returns a list starting with the first occurrence of *object*. Otherwise, it returns `nil`. The letter ‘`q`’ in `memq` says that it uses `eq` to compare *object* against the elements of the list. For example:

```
(memq 'b '(a b c b a))
     ⇒ (b c b a)
(memq '(2) '((1) (2)))    ; (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 75).

```
(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 '(2) 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*.[2]

Here is an example of an alist. The key `pine` is associated with the value `cones`; the key `oak` is associated with `acorns`; and the key `maple` is associated with `seeds`.

```
((pine . cones)
 (oak . acorns)
 (maple . seeds))
```

Both the values and the keys in an alist may be any Lisp objects. For example, in the following alist, the symbol `a` is associated with the number `1`, and the string `"b"` is associated with the *list* `(2 3)`, which is the CDR of the alist element:

```
((a . 1) ("b" 2 3))
```

Sometimes it is better to design an alist to store the associated value in the CAR of the CDR of the element. Here is an example of such an alist:

```
((rose red) (lily white) (buttercup yellow))
```

Here we regard `red` as the value associated with `rose`. One advantage of this kind of alist is that you can store other related information—even a list of other items—in the CDR of the CDR. One disadvantage is that you cannot use `rassq` (see below) to find the element containing a given value. When neither of these considerations is important, the choice is a matter of taste, as long as you are consistent about it for any given alist.

The same alist shown above could be regarded as having the associated value in the CDR of the element; the value associated with `rose` would be the list `(red)`.

Association lists are often used to record information that you might otherwise keep on a stack, since new associations may be added easily to the front of the list. When searching an association list for an association with a given key, the first one found is returned, if there is more than one.

In Emacs Lisp, it is *not* an error if an element of an association list is not a cons cell. The alist search functions simply ignore such elements. Many other versions of Lisp signal errors in such cases.

Note that property lists are similar to association lists in several respects. A property list behaves like an association list in which each key can occur only once. See Section 8.4 [Property Lists], page 106, 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)
```

[2] 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.

```
(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 53.

`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 221, vol. 2). 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.

6 Sequences, Arrays, and Vectors

The *sequence* type is the union of two other Lisp types: lists and arrays. In other words, any list is a sequence, and any array is a sequence. The common property that all sequences have is that each is an ordered collection of elements.

An *array* is a fixed-length object with a slot for each of its elements. All the elements are accessible in constant time. The four types of arrays are strings, vectors, char-tables and bool-vectors.

A list is a sequence of elements, but it is not a single primitive object; it is made of cons cells, one cell per element. Finding the *n*th element requires looking through *n* cons cells, so elements farther from the beginning of the list take longer to access. But it is possible to add elements to the list, or remove elements.

The following diagram shows the relationship between these types:

```
 _____________________________________________
|                                             |
|          Sequence                           |
|  ______   ________________________________  |
| |      | |                                | |
| | List | |             Array              | |
| |      | |    ________       ________     | |
| |______| |   |        |     |        |    | |
|          |   | Vector |     | String |    | |
|          |   |________|     |________|    | |
|          |  ____________   _____________  | |
|          | |            | |             | | |
|          | | Char-table | | Bool-vector | | |
|          | |____________| |_____________| | |
|          |________________________________| |
|_____________________________________________|
```

6.1 Sequences

This section describes functions that accept any kind of sequence.

`sequencep` *object* [Function]

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

`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 67, 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 182, vol. 2.

If you need to compute the width of a string on display, you should use `string-width` (see Section 38.10 [Width], page 323, vol. 2), not `length`, since `length` only counts the number of characters, but does not account for the display width of each character.

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

`copy-sequence` *sequence* [Function]

This 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 156, vol. 2.

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 68, `concat` in Section 4.3 [Creating Strings], page 49, and `vconcat` in Section 6.5 [Vector Functions], page 91, 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, the time to access an element of a list is proportional to the position of that element in the list.

Emacs defines four types of array, all one-dimensional: *strings* (see Section 2.3.8 [String Type], page 18), *vectors* (see Section 2.3.9 [Vector Type], page 20), *bool-vectors* (see Section 2.3.11 [Bool-Vector Type], page 21), and *char-tables* (see Section 2.3.10 [Char-Table Type], page 20). Vectors and char-tables can hold elements of any type, but strings can only hold characters, and bool-vectors can only hold `t` and `nil`.

All four kinds of array share these characteristics:

- The first element of an array has index zero, the second element has index 1, and so on. This is called *zero-origin* indexing. For example, an array of four elements has indices 0, 1, 2, and 3.
- The length of the array is fixed once you create it; you cannot change the length of an existing array.
- For purposes of evaluation, the array is a constant—i.e., it evaluates to itself.
- The elements of an array may be referenced or changed with the functions `aref` and `aset`, respectively (see Section 6.3 [Array Functions], page 89).

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 156, vol. 2.
- 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 48.

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 21.8.1 [Key Sequence Input], page 342.

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 86.

`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 86.

6.4 Vectors

A *vector* is a general-purpose array whose elements can be any Lisp objects. (By contrast, the elements of a string can only be characters. See Chapter 4 [Strings and Characters], page 48.) Vectors are used in Emacs for many purposes: as key sequences (see Section 22.1 [Key Sequences], page 360), as symbol-lookup tables (see Section 8.3 [Creating Symbols], page 104), as part of the representation of a byte-compiled function (see Chapter 16 [Byte Compilation], page 223), and more.

Like other arrays, vectors use zero-origin indexing: the first element has index 0.

Vectors are printed with square brackets surrounding the elements. Thus, a vector whose elements are the symbols `a`, `b` and `a` is printed as `[a b a]`. You can write vectors in the same way in Lisp input.

A vector, like a string or a number, is considered a constant for evaluation: the result of evaluating it is the same vector. This does not evaluate or even examine the elements of the vector. See Section 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 *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 229.

For other concatenation functions, see `mapconcat` in Section 12.6 [Mapping Functions], page 172, `concat` in Section 4.3 [Creating Strings], page 49, and `append` in Section 5.4 [Building Lists], page 68.

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. Like vectors, char-tables are constants when evaluated, and can hold elements of any type.

Each char-table has a *subtype*, a symbol, which serves two purposes:

- The subtype provides an easy way to tell what the char-table is for. For instance, display tables are char-tables with `display-table` as the subtype, and syntax tables are char-tables with `syntax-table` as the subtype. The subtype can be queried using the function `char-table-subtype`, described below.
- The subtype controls the number of *extra slots* in the char-table. This number is specified by the subtype's `char-table-extra-slots` symbol property, which should be an integer between 0 and 10. If the subtype has no such symbol property, the char-table has no extra slots. See Section 8.4 [Property Lists], page 106, for information about symbol properties.

A char-table can have a *parent*, which is another char-table. If it does, then whenever the char-table specifies `nil` for a particular character *c*, it inherits the value specified in the parent. In other words, `(aref char-table c)` returns the value from the parent of *char-table* if *char-table* itself specifies `nil`.

A char-table can also have a *default value*. If so, then `(aref char-table c)` returns the default value whenever the char-table does not specify any other non-`nil` value.

`make-char-table` *subtype* **&optional** *init* [Function]

Return a newly-created char-table, with subtype *subtype* (a symbol). Each element is initialized to *init*, which defaults to `nil`. You cannot alter the subtype of a char-table after the char-table is created.

There is no argument to specify the length of the char-table, because all char-tables have room for any valid character code as an index.

If *subtype* has the `char-table-extra-slots` symbol property, that specifies the number of extra slots in the char-table. This should be an integer between 0 and 10; otherwise, `make-char-table` raises an error. If *subtype* has no `char-table-extra-slots` symbol property (see Section 8.4 [Property Lists], page 106), the char-table has no extra slots.

`char-table-p` *object* [Function]
: This function returns `t` if *object* is a char-table, and `nil` otherwise.

`char-table-subtype` *char-table* [Function]
: This function returns the subtype symbol of *char-table*.

There is no special function to access default values in a char-table. To do that, use `char-table-range` (see below).

`char-table-parent` *char-table* [Function]
: This function returns the parent of *char-table*. The parent is always either `nil` or another char-table.

`set-char-table-parent` *char-table new-parent* [Function]
: This function sets the parent of *char-table* to *new-parent*.

`char-table-extra-slot` *char-table n* [Function]
: This function returns the contents of extra slot *n* 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).

`(`*`from`*` . `*`to`*`)`
A cons cell refers to all the characters in the inclusive range '`[`*`from`*`..`*`to`*`]`'.

`set-char-table-range` *char-table range value* [Function]
: This function sets the value in *char-table* for a range of characters *range*. Here are the possibilities for *range*:

`nil` Refers to the default value.

`t` Refers to the whole range of character codes.

char Refers to the element for character *char* (supposing *char* is a valid character code).

`(`*`from`*` . `*`to`*`)`
A cons cell refers to all the characters in the inclusive range '`[`*`from`*`..`*`to`*`]`'.

`map-char-table` *function char-table* [Function]

This function calls its argument *function* for each element of *char-table* that has a non-`nil` value. The call to *function* is with two arguments, a key and a value. The key is a possible *range* argument for `char-table-range`—either a valid character or a cons cell `(from . to)`, specifying a range of characters that share the same value. The value is what `(char-table-range char-table key)` returns.

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

The return value is always `nil`; to make calls to `map-char-table` useful, *function* should have side effects. For example, here is how to examine the elements of the syntax table:

```
(let (accumulator)
   (map-char-table
    #'(lambda (key value)
        (setq accumulator
              (cons (list
                     (if (consp key)
                         (list (car key) (cdr key))
                       key)
                     value)
                    accumulator)))
    (syntax-table))
   accumulator)
⇒
(((2597602 4194303) (2)) ((2597523 2597601) (3))
 ... (65379 (5 . 65378)) (65378 (4 . 65379)) (65377 (1))
 ... (12 (0)) (11 (3)) (10 (12)) (9 (0)) ((0 8) (3)))
```

6.7 Bool-vectors

A bool-vector is much like a vector, except that it stores only the values `t` and `nil`. If you try to store any non-`nil` value into an element of the bool-vector, the effect is to store `t` there. As with all arrays, bool-vector indices start from 0, and the length cannot be changed once the bool-vector is created. Bool-vectors are constants when evaluated.

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.

6.8 Managing a Fixed-Size Ring of Objects

A *ring* is a fixed-size data structure that supports insertion, deletion, rotation, and modulo-indexed reference and traversal. An efficient ring data structure is implemented by the `ring` package. It provides the functions listed in this section.

Note that several "rings" in Emacs, like the kill ring and the mark ring, are actually implemented as simple lists, *not* using the `ring` package; thus the following functions won't work on them.

`make-ring` *size* [Function]
: This returns a new ring capable of holding *size* objects. *size* should be an integer.

`ring-p` *object* [Function]
: This returns `t` if *object* is a ring, `nil` otherwise.

`ring-size` *ring* [Function]
: This returns the maximum capacity of the *ring*.

`ring-length` *ring* [Function]
: This returns the number of objects that *ring* currently contains. The value will never exceed that returned by `ring-size`.

`ring-elements` *ring* [Function]
: This returns a list of the objects in *ring*, in order, newest first.

`ring-copy` *ring* [Function]
: This returns a new ring which is a copy of *ring*. The new ring contains the same (`eq`) objects as *ring*.

`ring-empty-p` *ring* [Function]
: This returns `t` if *ring* is empty, `nil` otherwise.

The newest element in the ring always has index 0. Higher indices correspond to older elements. Indices are computed modulo the ring length. Index −1 corresponds to the oldest element, −2 to the next-oldest, and so forth.

`ring-ref` *ring index* [Function]
: This returns the object in *ring* found at index *index*. *index* may be negative or greater than the ring length. If *ring* is empty, `ring-ref` signals an error.

`ring-insert` *ring object* [Function]

This inserts *object* into *ring*, making it the newest element, and returns *object*.

If the ring is full, insertion removes the oldest element to make room for the new element.

`ring-remove` *ring* **&optional** *index* [Function]

Remove an object from *ring*, and return that object. The argument *index* specifies which item to remove; if it is `nil`, that means to remove the oldest item. If *ring* is empty, `ring-remove` signals an error.

`ring-insert-at-beginning` *ring object* [Function]

This inserts *object* into *ring*, treating it as the oldest element. The return value is not significant.

If the ring is full, this function removes the newest element to make room for the inserted element.

If you are careful not to exceed the ring size, you can use the ring as a first-in-first-out queue. For example:

```
(let ((fifo (make-ring 5)))
  (mapc (lambda (obj) (ring-insert fifo obj))
        '(0 one "two"))
  (list (ring-remove fifo) t
        (ring-remove fifo) t
        (ring-remove fifo)))
     ⇒ (0 t one t "two")
```

7 Hash Tables

A hash table is a very fast kind of lookup table, somewhat like an alist (see Section 5.8 [Association Lists], page 82) in that it maps keys to corresponding values. It differs from an alist in these ways:

- Lookup in a hash table is extremely fast for large tables—in fact, the time required is essentially *independent* of how many elements are stored in the table. For smaller tables (a few tens of elements) alists may still be faster because hash tables have a more-or-less constant overhead.
- The correspondences in a hash table are in no particular order.
- There is no way to share structure between two hash tables, the way two alists can share a common tail.

Emacs Lisp provides a general-purpose hash table data type, along with a series of functions for operating on them. Hash tables have a special printed representation, which consists of '`#s`' followed by a list specifying the hash table properties and contents. See Section 7.1 [Creating Hash], page 97. (Note that the term "hash notation", which refers to the initial '`#`' character used in the printed representations of objects with no read representation, has nothing to do with the term "hash table". See Section 2.1 [Printed Representation], page 8.)

Obarrays are also a kind of hash table, but they are a different type of object and are used only for recording interned symbols (see Section 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 100) 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.

You can also create a new hash table using the printed representation for hash tables. The Lisp reader can read this printed representation, provided each element in the specified hash table has a valid read syntax (see Section 2.1 [Printed Representation], page 8). For instance, the following specifies a new hash table containing the keys `key1` and `key2` (both symbols) associated with `val1` (a symbol) and `300` (a number) respectively.

```
#s(hash-table size 30 data (key1 val1 key2 300))
```

The printed representation for a hash table consists of '`#s`' followed by a list beginning with '`hash-table`'. The rest of the list should consist of zero or more property-value pairs specifying the hash table's properties and initial contents. The properties and values are read literally. Valid property names are `size`, `test`, `weakness`, `rehash-size`, `rehash-threshold`, and `data`. The `data` property should be a list of key-value pairs for the initial contents; the other properties have the same meanings as the matching `make-hash-table` keywords (`:size`, `:test`, etc.), described above.

Note that you cannot specify a hash table whose initial contents include objects that have no read syntax, such as buffers and frames. Such objects may be added to the hash table after it is created.

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 137, and Chapter 12 [Functions], page 163. 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 symbol’s name.
Value	The symbol’s current value as a variable.
Function	The symbol’s function definition. It can also hold a symbol, a keymap, or a keyboard macro.
Property list	The symbol’s property list.

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

The print name cell holds the string that is the name of a symbol. Since symbols are represented textually by their names, it is important not to have two symbols with the same name. The Lisp reader ensures this: every time it reads a symbol, it looks for an existing symbol with the specified name before it creates a new one. To get a symbol’s name, use the function `symbol-name` (see Section 8.3 [Creating Symbols], page 104).

The value cell holds a symbol’s value as a variable, which is what you get if the symbol itself is evaluated as a Lisp expression. See Chapter 11 [Variables], page 137, for details about how values are set and retrieved, including complications such as *local bindings* and *scoping rules*. Most symbols can have any Lisp object as a value, but certain special symbols have values that cannot be changed; these include `nil` and `t`, and any symbol whose name starts with ‘`:`’ (those are called *keywords*). See Section 11.2 [Constant Variables], page 137.

The function cell holds a symbol’s function definition. Often, we refer to “the function `foo`” when we really mean the function stored in the function cell of `foo`; we make the distinction explicit only when necessary. Typically, the function cell is used to hold a function (see Chapter 12 [Functions], page 163) or a macro (see Chapter 13 [Macros], page 181). However, it can also be used to hold a symbol (see Section 9.1.4 [Function Indirection], page 112), keyboard macro (see Section 21.16 [Keyboard Macros], page 358), keymap (see Chapter 22 [Keymaps], page 360), or autoload object (see Section 9.1.8 [Autoloading], page 116). To get the contents of a symbol’s function cell, use the function `symbol-function` (see Section 12.8 [Function Cells], page 175).

The property list cell normally should hold a correctly formatted property list. To get a symbol's property list, use the function `symbol-plist`. See Section 8.4 [Property Lists], page 106.

The function cell or the value cell may be *void*, which means that the cell does not reference any object. (This is not the same thing as holding the symbol `void`, nor the same as holding the symbol `nil`.) Examining a function or value cell that is void results in an error, such as '`Symbol's value as variable is void`'.

Because each symbol has separate value and function cells, variables names and function names do not conflict. For example, the symbol `buffer-file-name` has a value (the name of the file being visited in the current buffer) as well as a function definition (a primitive function that returns the name of the file):

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

8.2 Defining Symbols

A *definition* is a special kind of Lisp expression that announces your intention to use a symbol in a particular way. It typically specifies a value or meaning for the symbol for one kind of use, plus documentation for its meaning when used in this way. Thus, when you define a symbol as a variable, you can supply an initial value for the variable, plus documentation for the variable.

`defvar` and `defconst` are special forms that define a symbol as a *global variable*—a variable that can be accessed at any point in a Lisp program. See Chapter 11 [Variables], page 137, for details about variables. To define a customizable variable, use the `defcustom` macro, which also calls `defvar` as a subroutine (see Chapter 14 [Customization], page 190).

In principle, you can assign a variable value to any symbol with `setq`, whether not it has first been defined as a variable. However, you ought to write a variable definition for each global variable that you want to use; otherwise, your Lisp program may not act correctly if it is evaluated with lexical scoping enabled (see Section 11.9 [Variable Scoping], page 146).

`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 163.

`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 181.

As previously noted, Emacs Lisp allows the same symbol to be defined both as a variable (e.g. with `defvar`) and as a function or macro (e.g. with `defun`). Such definitions do not conflict.

These definition also act as guides for programming tools. For example, the `C-h f` and `C-h v` commands create help buffers containing links to the relevant variable, function, or macro definitions. See Section "Name Help" in *The GNU Emacs Manual*.

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:** Unlike Common Lisp, Emacs Lisp does not provide for interning a single symbol 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 452, for another example using `mapatoms`.

`unintern` *symbol obarray* [Function]

This function deletes *symbol* from the obarray *obarray*. If `symbol` is not actually in the obarray, `unintern` does nothing. If *obarray* is `nil`, the current obarray is used.

If you provide a string instead of a symbol as *symbol*, it stands for a symbol name. Then `unintern` deletes the symbol (if any) in the obarray which has that name. If there is no such symbol, `unintern` does nothing.

If `unintern` does delete a symbol, it returns `t`. Otherwise it returns `nil`.

8.4 Property Lists

A *property list* (*plist* for short) is a list of paired elements. Each of the pairs associates a property name (usually a symbol) with a property or value.

Every symbol has a cell that stores a property list (see Section 8.1 [Symbol Components], page 102). This property list is used to record information about the symbol, such as its variable documentation and the name of the file where it was defined.

Property lists can also be used in other contexts. For instance, you can assign property lists to character positions in a string or buffer. See Section 32.19 [Text Properties], page 156, vol. 2.

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 82) are very similar to property lists. In contrast to association lists, the order of the pairs in the property list is not significant since the property names must be distinct.

Property lists are better than association lists for attaching information to various Lisp function names or variables. If your program keeps all such information in one association list, it will typically need to search that entire list each time it checks for an association for a particular Lisp function name or variable, which could be slow. By contrast, if you keep the same information in the property lists of the function names or variables themselves, each search will scan only the length of one property list, which is usually short. This is why the documentation for a variable is recorded in a property named `variable-documentation`. The byte compiler likewise uses properties to record those functions needing special treatment.

However, association lists have their own advantages. Depending on your application, it may be faster to add an association to the front of an association list than to update a property. All properties for a symbol are stored in the same property list, so there is a possibility of a conflict between different uses of a property name. (For this reason, it is a good idea to choose property names that are probably unique, such as by beginning the property name with the program's usual name-prefix for variables and functions.) An association list may be used like a stack where associations are pushed on the front of the list and later discarded; this is not possible with a property list.

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 250, vol. 2).

`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 not stored in symbols:

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

This returns the value of the *property* property stored in the property list *plist*. It accepts a malformed *plist* argument. If *property* is not found in the *plist*, it returns `nil`. For example,

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

`plist-put` *plist property value* [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 a *form* or *expression*[1]. The fact that forms are data objects and not merely text is one of the fundamental differences between Lisp-like languages and typical programming languages. Any object can be evaluated, but in practice only numbers, symbols, lists and strings are evaluated very often.

In subsequent sections, we will describe the details of what evaluation means for each kind of form.

It is very common to read a Lisp form and then evaluate the form, but reading and evaluation are separate activities, and either can be performed alone. Reading per se does not evaluate anything; it converts the printed representation of a Lisp object to the object itself. It is up to the caller of `read` to specify whether this object is a form to be evaluated, or serves some entirely different purpose. See Section 19.3 [Input Functions], page 276.

Evaluation is a recursive process, and evaluating a form often involves evaluating parts within that form. For instance, when you evaluate a *function call* form such as `(car x)`, Emacs first evaluates the argument (the subform `x`). After evaluating the argument, Emacs *executes* the function (`car`), and if the function is written in Lisp, execution works by evaluating the *body* of the function (in this example, however, `car` is not a Lisp function; it is a primitive function implemented in C). See Chapter 12 [Functions], page 163, for more information about functions and function calls.

Evaluation takes place in a context called the *environment*, which consists of the current values and bindings of all Lisp variables (see Chapter 11 [Variables], page 137).[2] Whenever a form refers to a variable without creating a new binding for it, the variable evaluates to the value given by the current environment. Evaluating a form may also temporarily alter the environment by binding variables (see Section 11.3 [Local Variables], page 138).

Evaluating a form may also make changes that persist; these changes are called *side effects*. An example of a form that produces a side effect is `(setq foo 1)`.

Do not confuse evaluation with command key interpretation. The editor command loop translates keyboard input into a command (an interactively callable function) using the active keymaps, and then uses `call-interactively` to execute that command. Executing the command usually involves evaluation, if the command is written in Lisp; however, this step is not considered a part of command key interpretation. See Chapter 21 [Command Loop], page 315.

[1] It is sometimes also referred to as an *S-expression* or *sexp*, but we generally do not use this terminology in this manual.

[2] This definition of "environment" is specifically not intended to include all the data that can affect the result of a program.

9.1 Kinds of Forms

A Lisp object that is intended to be evaluated is called a *form* (or an *expression*). How Emacs evaluates a form depends on its data type. Emacs has three different kinds of form that are evaluated differently: symbols, lists, and "all other types". This section describes all three kinds, one by one, starting with the "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, evaluating a vector does not cause evaluation of the elements of the vector—it returns the same vector with its contents unchanged.

```
'123                  ; A number, shown without evaluation.
     ⇒ 123
123                   ; Evaluated as usual—result is the same.
     ⇒ 123
(eval '123)           ; Evaluated ``by hand''—result is the same.
     ⇒ 123
(eval (eval '123))    ; Evaluating twice changes nothing.
     ⇒ 123
```

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 the symbol has no value as a variable, the Lisp interpreter signals an error. For more information on the use of variables, see Chapter 11 [Variables], page 137.

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 137.

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 168, for more information about symbol function indirection.

One possible consequence of this process is an infinite loop, in the event that a symbol's function cell refers to the same symbol. 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, Emacs signals an `invalid-function` error.

The following example illustrates the symbol indirection process. We use `fset` to set the function cell of a symbol and `symbol-function` to get the function cell contents (see Section 12.8 [Function Cells], page 175). Specifically, we store the symbol `car` into the function cell of `first`, and the symbol `first` into the function cell of `erste`.

```
;; Build this function cell linkage:
;;   -------------       -----        -------        -------
;;  | #<subr car> | <-- | car |  <-- | first |  <-- | erste |
;;   -------------       -----        -------        -------
(symbol-function 'car)
     ⇒ #<subr car>
(fset 'first 'car)
     ⇒ car
(fset 'erste 'first)
     ⇒ first
(erste '(1 2 3))   ; Call the function referenced by erste.
     ⇒ 1
```

By contrast, the following example calls a function without any symbol function indirection, because the first element is an anonymous Lisp function, not a symbol.

```
((lambda (arg) (erste arg))
 '(1 2 3))
     ⇒ 1
```

Executing the function itself evaluates its body; this does involve symbol function indirection when calling `erste`.

This form is rarely used and is now deprecated. Instead, you should write it as:

```
(funcall (lambda (arg) (erste arg))
         '(1 2 3))
```

or just

```
(let ((arg '(1 2 3))) (erste arg))
```

The built-in function `indirect-function` provides an easy way to perform symbol function indirection explicitly.

`indirect-function` *function* **&optional** *noerror* [Function]

This function returns the meaning of *function* as a function. If *function* is a symbol, then it finds *function*'s function definition and starts over with that value. If *function* is not a symbol, then it returns *function* itself.

This function 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 170). 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 165); 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 181, 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.

`and`	see Section 10.3 [Combining Conditions], page 123
`catch`	see Section 10.5.1 [Catch and Throw], page 126
`cond`	see Section 10.2 [Conditionals], page 121
`condition-case`	see Section 10.5.3.3 [Handling Errors], page 130
`defconst`	see Section 11.5 [Defining Variables], page 141
`defmacro`	see Section 13.4 [Defining Macros], page 183
`defun`	see Section 12.4 [Defining Functions], page 169
`defvar`	see Section 11.5 [Defining Variables], page 141

`function` see Section 12.7 [Anonymous Functions], page 174

`if` see Section 10.2 [Conditionals], page 121

`interactive`
see Section 21.3 [Interactive Call], page 321

`let`
`let*` see Section 11.3 [Local Variables], page 138

`or` see Section 10.3 [Combining Conditions], page 123

`prog1`
`prog2`
`progn` see Section 10.1 [Sequencing], page 120

`quote` see Section 9.2 [Quoting], page 116

`save-current-buffer`
see Section 27.2 [Current Buffer], page 1, vol. 2

`save-excursion`
see Section 30.3 [Excursions], page 108, vol. 2

`save-restriction`
see Section 30.4 [Narrowing], page 109, vol. 2

`save-window-excursion`
see Section 28.23 [Window Configurations], page 60, vol. 2

`setq` see Section 11.8 [Setting Variables], page 145

`setq-default`
see Section 11.10.2 [Creating Buffer-Local], page 152

`track-mouse`
see Section 29.13 [Mouse Tracking], page 87, vol. 2

`unwind-protect`
see Section 10.5 [Nonlocal Exits], page 126

`while` see Section 10.4 [Iteration], page 124

`with-output-to-temp-buffer`
see Section 38.8 [Temporary Displays], page 313, vol. 2

> **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. The way to arrange for an autoload object to appear as a symbol's function definition is described in Section 15.5 [Autoload], page 213.

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 174), which causes an anonymous lambda expression written in Lisp to be compiled, and '`` ` ``' (see Section 9.3 [Backquote], page 116), which is used to quote only part of a list, while computing and substituting other parts.

9.3 Backquote

Backquote constructs allow you to quote a list, but selectively evaluate elements of that list. In the simplest case, it is identical to the special form `quote` For example, these two forms yield identical results:

```
`(a list of (+ 2 3) elements)
     ⇒ (a list of (+ 2 3) elements)
```

```
'(a list of (+ 2 3) elements)
     ⇒ (a list of (+ 2 3) elements)
```

The special marker '`,`' inside of the argument to backquote indicates a value that isn't constant. The Emacs Lisp evaluator evaluates the argument of '`,`', and puts the value in the list structure:

```
`(a list of ,(+ 2 3) elements)
     ⇒ (a list of 5 elements)
```

Substitution with '`,`' is allowed at deeper levels of the list structure also. For example:

```
`(1 2 (3 ,(+ 4 5)))
     ⇒ (1 2 (3 9))
```

You can also *splice* an evaluated value into the resulting list, using the special marker '`,@`'. The elements of the spliced list become elements at the same level as the other elements of the resulting list. The equivalent code without using '`` ` ``' is often unreadable. Here are some examples:

```
(setq some-list '(2 3))
     ⇒ (2 3)
(cons 1 (append some-list '(4) some-list))
     ⇒ (1 2 3 4 2 3)
`(1 ,@some-list 4 ,@some-list)
     ⇒ (1 2 3 4 2 3)

(setq list '(hack foo bar))
     ⇒ (hack foo bar)
(cons 'use
  (cons 'the
    (cons 'words (append (cdr list) '(as elements)))))
     ⇒ (use the words foo bar as elements)
`(use the words ,@(cdr list) as elements)
     ⇒ (use the words foo bar as elements)
```

9.4 Eval

Most often, forms are evaluated automatically, by virtue of their occurrence in a program being run. On rare occasions, you may need to write code that evaluates a form that is computed at run time, such as after reading a form from text being edited or getting one from a property list. On these occasions, use the `eval` function. Often `eval` is not needed and something else should be used instead. For example, to get the value of a variable, while `eval` works, `symbol-value` is preferable; or rather than store expressions in a property list that then need to go through `eval`, it is better to store functions instead that are then passed to `funcall`.

The functions and variables described in this section evaluate forms, specify limits to the evaluation process, or record recently returned values. Loading a file also does evaluation (see Chapter 15 [Loading], page 209).

It is generally cleaner and more flexible to store a function in a data structure, and call it with `funcall` or `apply`, than to store an expression in the data structure and evaluate it. Using functions provides the ability to pass information to them as arguments.

`eval` *form* **&optional** *lexical* [Function]

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

The argument *lexical*, if non-`nil`, means to evaluate *form* using lexical scoping rules for variables, instead of the default dynamic scoping rules. See Section 11.9.3 [Lexical Binding], page 148.

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 279), 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 277.

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 211) 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 220), and defaults to `buffer-file-name` (see Section 27.4 [Buffer File Name], page 5, vol. 2). 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` [User Option]

This variable defines the maximum depth allowed in calls to `eval`, `apply`, and `funcall` before an error is signaled (with error message `"Lisp nesting exceeds max-lisp-eval-depth"`).

This limit, with the associated error when it is exceeded, is 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 400. 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 139.

`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 a set 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 181).

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 constructs likewise evaluate a series of forms but return different values:

`prog1` *form1 forms...* [Special Form]

This special form evaluates *form1* and all of the *forms*, in textual order, returning the result of *form1*.

```
(prog1 (print "The first form")
       (print "The second form")
       (print "The third form"))
     ⊣ "The first form"
     ⊣ "The second form"
     ⊣ "The third form"
⇒ "The first form"
```

Here is a way to remove the first element from a list in the variable `x`, then return the value of that former element:

```
(prog1 (car x) (setq x (cdr x)))
```

`prog2` *form1 form2 forms...* [Special Form]

This special form evaluates *form1*, *form2*, and all of the following *forms*, in textual order, returning the result of *form2*.

```
(prog2 (print "The first form")
       (print "The second form")
       (print "The third form"))
     ⊣ "The first form"
     ⊣ "The second form"
     ⊣ "The third form"
⇒ "The second form"
```

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 120.)

If *condition* has the value `nil`, and no *else-forms* are given, `if` returns `nil`.

`if` is a special form because the branch that is not selected is never evaluated—it is ignored. Thus, in this example, `true` is not printed because `print` is never called:

```
(if nil
    (print 'true)
  'very-false)
⇒ very-false
```

`when` *condition then-forms...* [Macro]

This is a variant of `if` where there are no *else-forms*, and possibly several *then-forms*. In particular,

```
(when condition a b c)
```

is entirely equivalent to

```
(if condition (progn a b c) nil)
```

`unless` *condition forms...* [Macro]

This is a variant of `if` where there is no *then-form*:

```
(unless condition a b c)
```

is entirely equivalent to

```
(if condition nil
   a b c)
```

`cond` *clause...* [Special Form]

`cond` chooses among an arbitrary number of alternatives. Each *clause* in the `cond` must be a list. The CAR of this list is the *condition*; the remaining elements, if any, the *body-forms*. Thus, a clause looks like this:

```
(condition body-forms...)
```

`cond` tries the clauses in textual order, by evaluating the *condition* of each clause. If the value of *condition* is non-`nil`, the clause "succeeds"; then `cond` evaluates its *body-forms*, and 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 126).

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 138). Likewise, `throw` restores the buffer and position saved by `save-excursion` (see Section 30.3 [Excursions], page 108, vol. 2), and the narrowing status saved by `save-restriction` and the window selection saved by `save-window-excursion` (see Section 28.23 [Window Configurations], page 60, vol. 2). It also runs any cleanups established with the `unwind-protect` special form when it exits that form (see Section 10.5.4 [Cleanups], page 135).

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 21.13 [Recursive Editing], page 355).

> **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 “goto”.) Here we compute `(foo i j)` for *i* and *j* varying from 0 to 9:

```
(defun search-foo ()
  (catch 'loop
    (let ((i 0))
      (while (< i 10)
        (let ((j 0))
          (while (< j 10)
            (if (foo i j)
                (throw 'loop (list i j)))
            (setq j (1+ j))))
        (setq i (1+ i))))))
```

If `foo` ever returns non-`nil`, we stop immediately and return a list of *i* and *j*. If `foo` always returns `nil`, the `catch` returns normally, and the value is `nil`, since that is the result of the `while`.

Here are two tricky examples, slightly different, showing two return points at once. First, two return points with the same tag, `hack`:

```
(defun catch2 (tag)
  (catch tag
    (throw 'hack 'yes)))
⇒ catch2
```

```
(catch 'hack
  (print (catch2 'hack))
  'no)
⊣ yes
⇒ no
```

Since both return points have tags that match the `throw`, it goes to the inner one, the one established in `catch2`. Therefore, `catch2` returns normally with value `yes`, and this value is printed. Finally the second body form in the outer `catch`, which is `'no`, is evaluated and returned from the outer `catch`.

Now let's change the argument given to `catch2`:

```
(catch 'hack
  (print (catch2 'quux))
  'no)
⇒ yes
```

We still have two return points, but this time only the outer one has the tag `hack`; the inner one has the tag `quux` instead. Therefore, `throw` makes the outer `catch` return the value `yes`. The function `print` is never called, and the body-form `'no` is never evaluated.

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 135.) 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 126.

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 130). 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 21.11 [Quitting], page 351.

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 57) 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 134, for a description of error symbols, error conditions and condition names.

If the error is not handled, the two arguments are used in printing the error message. Normally, this error message is provided by the `error-message` property of *error-symbol*. If *data* is non-`nil`, this is followed by a colon and a comma separated list of the unevaluated elements of *data*. For `error`, the error message is the CAR of *data* (that must be a string). Subcategories of `file-error` are handled specially.

The number and significance of the objects in *data* depends on *error-symbol*. For example, with a `wrong-type-argument` error, there should be two objects in the list: a predicate that describes the type that was expected, and the object that failed to fit that type.

Both *error-symbol* and *data* are available to any error handlers that handle the error: `condition-case` binds a local variable to a list of the form `(error-symbol . data)` (see Section 10.5.3.3 [Handling Errors], page 130).

The function `signal` never returns.

```
(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 243) 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. (You can use the macro `ignore-errors` for a simple case like this; see below.)

The `condition-case` construct is often used to trap errors that are predictable, such as failure to open a file in a call to `insert-file-contents`. It is also used to trap errors that are totally unpredictable, such as when the program evaluates an expression read from the user.

The second argument of `condition-case` is called the *protected form*. (In the example above, the protected form is a call to `delete-file`.) The error handlers go into effect when this form begins execution and are deactivated when this form returns. They remain in effect for all the intervening time. In particular, they are in effect during the execution of functions called by this form, in their subroutines, and so on. This is a good thing, since, strictly speaking, errors can be signaled only by Lisp primitives (including `signal` and `error`) called by the protected form, not by the protected form itself.

The arguments after the protected form are handlers. Each handler lists one or more *condition names* (which are symbols) to specify which errors it will handle. The error symbol specified when an error is signaled also defines a list of condition names. A handler applies to an error if they have any condition names in common. In the example above, there is one handler, and it specifies one condition name, `error`, which covers all errors.

The search for an applicable handler checks all the established handlers starting with the most recently established one. Thus, if two nested `condition-case` forms offer to handle the same error, the inner of the two gets to handle it.

If an error is handled by some `condition-case` form, this ordinarily prevents the debugger from being run, even if `debug-on-error` says this error should invoke the debugger.

If you want to be able to debug errors that are caught by a `condition-case`, set the variable `debug-on-signal` to a non-`nil` value. You can also specify that a particular handler should let the debugger run first, by writing `debug` among the conditions, like this:

```
(condition-case nil
    (delete-file filename)
  ((debug error) nil))
```

The effect of `debug` here is only to prevent `condition-case` from suppressing the call to the debugger. Any given error will invoke the debugger only if `debug-on-error` and the other usual filtering mechanisms say it should. See Section 18.1.1 [Error Debugging], page 243.

`condition-case-unless-debug` *var protected-form handlers...* [Macro]

> The macro `condition-case-unless-debug` provides another way to handle debugging of such forms. It behaves exactly like `condition-case`, unless the variable `debug-on-error` is non-`nil`, in which case it does not handle any errors at all.

Once Emacs decides that a certain handler handles the error, it returns control to that handler. To do so, Emacs unbinds all variable bindings made by binding constructs that are being exited, and executes the cleanups of all `unwind-protect` forms that are being exited. Once control arrives at the handler, the body of the handler executes normally.

After execution of the handler body, execution returns from the `condition-case` form. Because the protected form is exited completely before execution of the handler, the handler cannot resume execution at the point of the error, nor can it examine variable bindings that were made within the protected form. All it can do is clean up and proceed.

Error signaling and handling have some resemblance to `throw` and `catch` (see Section 10.5.1 [Catch and Throw], page 126), 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 (which can include `debug` to allow the debugger to run before the handler); *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 134). 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.

Sometimes it is necessary to re-throw a signal caught by `condition-case`, for some outer-level handler to catch. Here's how to do that:

```
  (signal (car err) (cdr err))
```

where `err` is the error description variable, the first argument to `condition-case` whose error condition you want to re-throw. See [Definition of signal], page 129.

`error-message-string` *error-descriptor* [Function]

This function returns the error message string for a given error descriptor. It is useful if you want to handle an error by printing the usual error message for that error. See [Definition of signal], page 129.

Here is an example of using `condition-case` to handle the error that results from dividing by zero. The handler displays the error message (but without a beep), then returns a very large number.

```
(defun safe-divide (dividend divisor)
  (condition-case err
      ;; Protected form.
      (/ dividend divisor)
    ;; The handler.
    (arith-error                        ; Condition.
     ;; Display the usual message for this error.
     (message "%s" (error-message-string err))
     1000000)))
⇒ safe-divide

(safe-divide 5 0)
     ⊣ Arithmetic error: (arith-error)
⇒ 1000000
```

The handler specifies condition name `arith-error` so that it will handle only division-by-zero errors. Other kinds of errors will not be handled (by this `condition-case`). Thus:

```
(safe-divide nil 3)
     error→ Wrong type argument: number-or-marker-p, nil
```

Here is a `condition-case` that catches all kinds of errors, including those from `error`:

```
(setq baz 34)
     ⇒ 34

(condition-case err
    (if (eq baz 35)
        t
      ;; This is a call to the function error.
      (error "Rats!  The variable %s was %s, not 35" 'baz baz))
  ;; This is the handler; it is not a form.
  (error (princ (format "The error was: %s" err))
         2))
⊣ The error was: (error "Rats!  The variable baz was 34, not 35")
⇒ 2
```

`ignore-errors` *body...* [Macro]

This construct executes *body*, ignoring any errors that occur during its execution. If the execution is without error, `ignore-errors` returns the value of the last form in *body*; otherwise, it returns `nil`.

Here's the example at the beginning of this subsection rewritten using `ignore-errors`:

```
(ignore-errors
 (delete-file filename))
```

`with-demoted-errors` *body...* [Macro]

This macro is like a milder version of `ignore-errors`. Rather than suppressing errors altogether, it converts them into messages. Use this form around code that is not expected to signal errors, but should be robust if one does occur. Note that this macro uses `condition-case-unless-debug` rather than `condition-case`.

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 129.

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 129) 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 477, vol. 2, for a list of the main 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 32.26 [Atomic Changes], page 179, vol. 2.)

`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 139).

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

```
(let ((buffer (get-buffer-create " *temp*")))
  (with-current-buffer buffer
    (unwind-protect
        body-form
      (kill-buffer buffer))))
```

You might think that we could just as well write `(kill-buffer (current-buffer))` and dispense with the variable `buffer`. However, the way shown above is safer, if *body-form* happens to get an error after switching to a different buffer! (Alternatively, you could write a `save-current-buffer` around *body-form*, to ensure that the temporary buffer becomes current again in time to kill it.)

Emacs includes a standard macro called `with-temp-buffer` which expands into more or less the code shown above (see [Current Buffer], page 3, vol. 2). 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 257, vol. 2) 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. In Lisp, each variable is represented by a Lisp symbol (see Chapter 8 [Symbols], page 102). The variable name is simply the symbol's name, and the variable's value is stored in the symbol's value cell[1]. See Section 8.1 [Symbol Components], page 102. In Emacs Lisp, the use of a symbol as a variable is independent of its use as a function name.

As previously noted in this manual, a Lisp program is represented primarily by Lisp objects, and only secondarily as text. The textual form of a Lisp program is given by the read syntax of the Lisp objects that constitute the program. Hence, the textual form of a variable in a Lisp program is written using the read syntax for the symbol representing the variable.

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` is a special form (see Section 9.1.7 [Special Forms], page 114); it does not evaluate its first argument, the name of the variable, but it does evaluate the second argument, the new value.

Once the variable has a value, you can refer to it by using the symbol itself as an expression. Thus,

```
x ⇒ (a b)
```

assuming the `setq` form shown above has already been executed.

If you do set the same variable again, the new value replaces the old one:

```
x
     ⇒ (a b)
(setq x 4)
     ⇒ 4
x
     ⇒ 4
```

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

[1] To be precise, under the default *dynamic binding* rules the value cell always holds the variable's current value, but this is not the case under *lexical binding* rules. See Section 11.9 [Variable Scoping], page 146, for details.

whose name starts with '`:`'), if it is interned in the standard obarray, except that setting such a symbol to itself is not an error.

```
nil ≡ 'nil
     ⇒ nil
(setq nil 500)
error  Attempt to set constant symbol: nil
```

`keywordp` *object* [Function]

function returns `t` if *object* is a symbol whose name starts with '`:`', interned in the standard obarray, and returns `nil` otherwise.

These constants are fundamentally different from the "constants" defined using the `defconst` special form (see Section 11.5 [Defining Variables], page 141). A `defconst` form serves to inform human readers that you do not intend to change the value of a variable, but Emacs does not raise an error if you actually change it.

11.3 Local Variables

Global variables have values that last until explicitly superseded with new values. Sometimes it is useful to give a variable a *local value*—a value that takes effect only within a certain part of a Lisp program. When a variable has a local value, we say that it is *locally bound* to that value, and that it is a *local variable*.

For example, when a function is called, its argument variables receive local values, which are the actual arguments supplied to the function call; these local bindings take effect within the body of the function. To take another example, the `let` special form explicitly establishes local bindings for specific variables, which take effect within the body of the `let` form.

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

Establishing a local binding saves away the variable's previous value (or lack of one). We say that the previous value is *shadowed*. Both global and local values may be shadowed. If a local binding is in effect, using `setq` on the local variable stores the specified value in the local binding. When that local binding is no longer in effect, the previously shadowed value (or lack of one) comes back.

A variable can have more than one local binding at a time (e.g. if there are nested `let` forms that bind the variable). The *current binding* is the local binding that is actually in effect. It determines the value returned by evaluating the variable symbol, and it is the binding acted on by `setq`.

For most purposes, you can think of the current binding as the "innermost" local binding, or the global binding if there is no local binding. To be more precise, a rule called the *scoping rule* determines where in a program a local binding takes effect. The default scoping rule in Emacs Lisp is called *dynamic scoping*, which simply states that the current binding at any given point in the execution of a program is the most recently-created binding for that variable that still exists. For details about dynamic scoping, and an alternative scoping rule called *lexical scoping*, See Section 11.9 [Variable Scoping], page 146.

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

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

This special form sets up local bindings for a certain set of variables, as specified by *bindings*, and then evaluates all of the *forms* in textual order. Its return value is the value of the last form in *forms*.

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

All of the *value-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 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 163).
- Macro calls (see Chapter 13 [Macros], page 181).
- `condition-case` (see Section 10.5.3 [Errors], page 128).

Variables can also have buffer-local bindings (see Section 11.10 [Buffer-Local Variables], page 150); a few variables have terminal-local bindings (see Section 29.2 [Multiple Terminals], page 67, vol. 2). These kinds of bindings work somewhat like ordinary local bindings, but they are localized depending on “where” you are in Emacs.

`max-specpdl-size` [User Option]

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 135) that are allowed before Emacs signals 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 119.

The default value is 1300. 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"

We say that a variable is void if its symbol has an unassigned value cell (see Section 8.1 [Symbol Components], page 102). Under Emacs Lisp's default dynamic binding rules (see Section 11.9 [Variable Scoping], page 146), the value cell stores the variable's current (local or global) value. Note that an unassigned value cell is *not* the same as having `nil` in the value cell. The symbol `nil` is a Lisp object and can be the value of a variable, just as any other object can be; but it is still a value. If a variable is void, trying to evaluate the variable signals a `void-variable` error rather than a value.

Under lexical binding rules, the value cell only holds the variable's global value, i.e. the value outside of any lexical binding construct. When a variable is lexically bound, the local value is determined by the lexical environment; the variable may have a local value if its symbol's value cell is unassigned.

makunbound *symbol* [Function]

This function empties out the value cell of *symbol*, making the variable void. It returns *symbol*.

If *symbol* has a dynamic local binding, `makunbound` voids the current binding, and this voidness lasts only as long as the local binding is in effect. Afterwards, the previously shadowed local or global binding is reexposed; then the variable will no longer be void, unless the reexposed binding is void too.

Here are some examples (assuming dynamic binding is in effect):

```
(setq x 1)               ; Put a value in the global binding.
     ⇒ 1
(let ((x 2))             ; Locally bind it.
  (makunbound 'x)        ; Void the local binding.
  x)
error  Symbol's value as variable is void: x
x                        ; The global binding is unchanged.
     ⇒ 1

(let ((x 2))             ; Locally bind it.
  (let ((x 3))           ; And again.
    (makunbound 'x)      ; Void the innermost-local binding.
    x))                  ; And refer: it's void.
error  Symbol's value as variable is void: x

(let ((x 2))
  (let ((x 3))
    (makunbound 'x))     ; Void inner binding, then remove it.
  x)                     ; Now outer let binding is visible.
     ⇒ 2
```

boundp *variable* [Function]

This function returns `t` if *variable* (a symbol) is not void, and `nil` if it is void.

Here are some examples (assuming dynamic binding is in effect):

```
(boundp 'abracadabra)          ; Starts out void.
     ⇒ nil
(let ((abracadabra 5))         ; Locally bind it.
  (boundp 'abracadabra))
     ⇒ t
(boundp 'abracadabra)          ; Still globally void.
     ⇒ nil
(setq abracadabra 5)           ; Make it globally nonvoid.
     ⇒ 5
(boundp 'abracadabra)
     ⇒ t
```

11.5 Defining Global Variables

A *variable definition* is a construct that announces your intention to use a symbol as a global variable. It uses the special forms `defvar` or `defconst`, which are documented below.

A variable definition serves three purposes. First, it informs people who read the code that the symbol is *intended* to be used a certain way (as a variable). Second, it informs the Lisp system of this, optionally supplying an initial value and a documentation string. Third, it provides information to programming tools such as `etags`, allowing them to find where the variable was defined.

The difference between `defconst` and `defvar` is mainly a matter of intent, serving to inform human readers of whether the value should ever change. Emacs Lisp does not actually prevent you from changing the value of a variable defined with `defconst`. One notable difference between the two forms is that `defconst` unconditionally initializes the variable, whereas `defvar` initializes it only if it is originally void.

To define a customizable variable, you should use `defcustom` (which calls `defvar` as a subroutine). See Chapter 14 [Customization], page 190.

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

This special form defines *symbol* as a variable. Note that *symbol* is not evaluated; the symbol to be defined should appear explicitly in the `defvar` form. The variable is marked as *special*, meaning that it should always be dynamically bound (see Section 11.9 [Variable Scoping], page 146).

If *symbol* is void and *value* is specified, `defvar` evaluates *value* 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 150.

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 is supplied, it specifies the documentation string for the variable (stored in the symbol's `variable-documentation` property). See Chapter 24 [Documentation], page 451.

Here are some examples. This form defines `foo` but does not initialize it:

```
(defvar foo)
     ⇒ foo
```

This example initializes the value of `bar` to `23`, and gives it a documentation string:

```
(defvar bar 23
  "The normal weight of a bar.")
     ⇒ bar
```

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

`defconst` *symbol value* [*doc-string*] [Special Form]

This special form defines *symbol* as a value and initializes it. It informs a person reading your code that *symbol* has a standard global value, established here, that should not be changed by the user or by other programs. Note that *symbol* is not evaluated; the symbol to be defined must appear explicitly in the `defconst`.

The `defconst` form, like `defvar`, marks the variable as *special*, meaning that it should always be dynamically bound (see Section 11.9 [Variable Scoping], page 146). In addition, it marks the variable as risky (see Section 11.11 [File Local Variables], page 156).

`defconst` always evaluates *value*, and sets the value of *symbol* to the result. If *symbol* does have a buffer-local binding in the current buffer, `defconst` sets the default value, not the buffer-local value. (But you should not be making buffer-local bindings for a symbol that is defined with `defconst`.)

An example of the use of `defconst` is Emacs's definition of `float-pi`—the mathematical constant *pi*, which ought not to be changed by anyone (attempts by the Indiana State Legislature notwithstanding). As the second form illustrates, however, `defconst` is only advisory.

```
(defconst float-pi 3.141592653589793 "The value of Pi.")
     ⇒ float-pi
(setq float-pi 3)
     ⇒ float-pi
float-pi
     ⇒ 3
```

Warning: If you use a `defconst` or `defvar` special form while the variable has a local binding (made with `let`, or a function argument), it sets the local binding rather than the global binding. This is not what you usually want. To prevent this, use these special forms at top level in a file, where normally no local binding is in effect, and make sure to load the file before making a local binding for the variable.

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 396).

'`...-function`'
: The value is a function.

'`...-functions`'
: The value is a list of functions.

'`...-form`'
: The value is a form (an expression).

'`...-forms`'
: The value is a list of forms (expressions).

'`...-predicate`'
: The value is a predicate—a function of one argument that returns non-`nil` for "good" arguments and `nil` for "bad" arguments.

'`...-flag`'
: The value is significant only as to whether it is `nil` or not. Since such variables often end up acquiring more values over time, this convention is not strongly recommended.

'`...-program`'
: The value is a program name.

'`...-command`'
: The value is a whole shell command.

'`...-switches`'
: The value specifies options for a command.

When you define a variable, always consider whether you should mark it as "safe" or "risky"; see Section 11.11 [File Local Variables], page 156.

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.

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.

Occasionally, you may want to reference a variable which is only determined at run time. In that case, you cannot specify the variable name in the text of the program. You can use the `symbol-value` function to extract the value.

`symbol-value` *symbol* [Function]

This function returns the value stored in *symbol*'s value cell. This is where the variable's current (dynamic) value is stored. If the variable has no local binding, this is simply its global value. If the variable is void, a `void-variable` error is signaled.

If the variable is lexically bound, the value reported by `symbol-value` is not necessarily the same as the variable's lexical value, which is determined by the lexical environment rather than the symbol's value cell. See Section 11.9 [Variable Scoping], page 146.

```
(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
```

11.8 Setting Variable Values

The usual way to change the value of a variable is with the special form `setq`. When you need to compute the choice of variable at run time, use the function `set`.

`setq` [*symbol form*]. . . [Special Form]

This special form is the most common method of changing a variable's value. Each *symbol* is given a new value, which is the result of evaluating the corresponding *form*. The current binding of the symbol is changed.

`setq` does not evaluate *symbol*; it sets the symbol that you write. We say that this argument is *automatically quoted*. The '`q`' in `setq` stands for "quoted".

The value of the `setq` form is the value of the last *form*.

```
(setq x (1+ 2))
     ⇒ 3
x                       ; x now has a global value.
     ⇒ 3
(let ((x 5))
  (setq x 6)            ; The local binding of x is set.
  x)
     ⇒ 6
x                       ; The global value is unchanged.
     ⇒ 3
```

Note that the first *form* is evaluated, then the first *symbol* is set, then the second *form* is evaluated, then the second *symbol* is set, and so on:

```
(setq x 10              ; Notice that x is set before
      y (1+ x))         ;    the value of y is computed.
     ⇒ 11
```

`set` *symbol value* [Function]

This function puts *value* in the value cell of *symbol*. Since it is a function rather than a special form, the expression written for *symbol* is evaluated to obtain the symbol to set. The return value is *value*.

When dynamic variable binding is in effect (the default), `set` has the same effect as `setq`, apart from the fact that `set` evaluates its *symbol* argument whereas `setq` does not. But when a variable is lexically bound, `set` affects its *dynamic* value, whereas `setq` affects its current (lexical) value. See Section 11.9 [Variable Scoping], page 146.

```
(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)
```

11.9 Scoping Rules for Variable Bindings

When you create a local binding for a variable, that binding takes effect only within a limited portion of the program (see Section 11.3 [Local Variables], page 138). This section describes exactly what this means.

Each local binding has a certain *scope* and *extent*. *Scope* refers to *where* in the textual source code the binding can be accessed. *Extent* refers to *when*, as the program is executing, the binding exists.

By default, the local bindings that Emacs creates are *dynamic bindings*. Such a binding has *indefinite scope*, meaning that any part of the program can potentially access the variable binding. It also has *dynamic extent*, meaning that the binding lasts only while the binding construct (such as the body of a `let` form) is being executed.

Emacs can optionally create *lexical bindings*. A lexical binding has *lexical scope*, meaning that any reference to the variable must be located textually within the binding construct. It also has *indefinite extent*, meaning that under some circumstances the binding can live on even after the binding construct has finished executing, by means of special objects called *closures*.

The following subsections describe dynamic binding and lexical binding in greater detail, and how to enable lexical binding in Emacs Lisp programs.

11.9.1 Dynamic Binding

By default, the local variable bindings made by Emacs are dynamic bindings. When a variable is dynamically bound, its current binding at any point in the execution of the Lisp program is simply the most recently-created dynamic local binding for that symbol, or the global binding if there is no such local binding.

Dynamic bindings have indefinite scope and dynamic extent, as shown by the following example:

```
(defvar x -99)   ; x receives an initial value of -99.

(defun getx ()
  x)             ; x is used “free” in this function.

(let ((x 1))     ; x is dynamically bound.
  (getx))
     ⇒ 1

;; After the let form finishes, x reverts to its
;; previous value, which is -99.

(getx)
     ⇒ -99
```

The function `getx` refers to `x`. This is a “free” reference, in the sense that there is no binding for `x` within that `defun` construct itself. When we call `getx` from within a `let` form in which `x` is (dynamically) bound, it retrieves the local value of `x` (i.e. 1). But when we call `getx` outside the `let` form, it retrieves the global value of `x` (i.e. -99).

Here is another example, which illustrates setting a dynamically bound variable using `setq`:

```
(defvar x -99)        ; x receives an initial value of -99.

(defun addx ()
  (setq x (1+ x)))    ; Add 1 to x and return its new value.

(let ((x 1))
  (addx)
  (addx))
     ⇒ 3              ; The two addx calls add to x twice.

;; After the let form finishes, x reverts to its
;; previous value, which is -99.

(addx)
     ⇒ -98
```

Dynamic binding is implemented in Emacs Lisp in a simple way. Each symbol has a value cell, which specifies its current dynamic value (or absence of value). See Section 8.1 [Symbol Components], page 102. When a symbol is given a dynamic local binding, Emacs records the contents of the value cell (or absence thereof) in a stack, and stores the new local value in the value cell. When the binding construct finishes executing, Emacs pops the old value off the stack, and puts it in the value cell.

11.9.2 Proper Use of Dynamic Binding

Dynamic binding is a powerful feature, as it allows programs to refer to variables that are not defined within their local textual scope. However, if used without restraint, this can also make programs hard to understand. There are two clean ways to use this technique:

- If a variable has no global definition, use it as a local variable only within a binding construct, e.g. the body of the `let` form where the variable was bound, or the body of the function for an argument variable. If this convention is followed consistently throughout a program, the value of the variable will not affect, nor be affected by, any uses of the same variable symbol elsewhere in the program.
- Otherwise, define the variable with `defvar`, `defconst`, or `defcustom`. See Section 11.5 [Defining Variables], page 141. Usually, the definition should be at top-level in an Emacs Lisp file. As far as possible, it should include a documentation string which explains the meaning and purpose of the variable. You should also choose the variable's name to avoid name conflicts (see Section D.1 [Coding Conventions], page 444, vol. 2).

  Then you can bind the variable anywhere in a program, knowing reliably what the effect will be. Wherever you encounter the variable, it will be easy to refer back to the definition, e.g. via the *C-h v* command (provided the variable definition has been loaded into Emacs). See Section "Name Help" in *The GNU Emacs Manual*.

  For example, it is common to use local bindings for customizable variables like `case-fold-search`:

  ```
  (defun search-for-abc ()
    "Search for the string \"abc\", ignoring case differences."
    (let ((case-fold-search nil))
      (re-search-forward "abc")))
  ```

11.9.3 Lexical Binding

Optionally, you can create lexical bindings in Emacs Lisp. A lexically bound variable has *lexical scope*, meaning that any reference to the variable must be located textually within the binding construct.

Here is an example (see the next subsection, for how to actually enable lexical binding):

```
(let ((x 1))    ; x is lexically bound.
  (+ x 3))
     ⇒ 4

(defun getx ()
  x)            ; x is used “free” in this function.

(let ((x 1))    ; x is lexically bound.
  (getx))
error  Symbol's value as variable is void: x
```

Here, the variable `x` has no global value. When it is lexically bound within a `let` form, it can be used in the textual confines of that `let` form. But it can *not* be used from within a `getx` function called from the `let` form, since the function definition of `getx` occurs outside the `let` form itself.

Here is how lexical binding works. Each binding construct defines a *lexical environment*, specifying the symbols that are bound within the construct and their local values. When the Lisp evaluator wants the current value of a variable, it looks first in the lexical environment; if the variable is not specified in there, it looks in the symbol's value cell, where the dynamic value is stored.

Lexical bindings have indefinite extent. Even after a binding construct has finished executing, its lexical environment can be "kept around" in Lisp objects called *closures*. A closure is created when you create a named or anonymous function with lexical binding enabled. See Section 12.9 [Closures], page 176, for details.

When a closure is called as a function, any lexical variable references within its definition use the retained lexical environment. Here is an example:

```
(defvar my-ticker nil)   ; We will use this dynamically bound
                         ; variable to store a closure.

(let ((x 0))             ; x is lexically bound.
  (setq my-ticker (lambda ()
                    (setq x (1+ x)))))
    ⇒ (closure ((x . 0) t) ()
          (1+ x))

(funcall my-ticker)
    ⇒ 1

(funcall my-ticker)
    ⇒ 2

(funcall my-ticker)
    ⇒ 3

x                        ; Note that x has no global value.
error  Symbol's value as variable is void: x
```

The `let` binding defines a lexical environment in which the variable `x` is locally bound to 0. Within this binding construct, we define a lambda expression which increments `x` by one and returns the incremented value. This lambda expression is automatically turned into a closure, in which the lexical environment lives on even after the `let` binding construct has exited. Each time we evaluate the closure, it increments `x`, using the binding of `x` in that lexical environment.

Note that functions like `symbol-value`, `boundp`, and `set` only retrieve or modify a variable's dynamic binding (i.e. the contents of its symbol's value cell). Also, the code in the body of a `defun` or `defmacro` cannot refer to surrounding lexical variables.

Currently, lexical binding is not much used within the Emacs sources. However, we expect its importance to increase in the future. Lexical binding opens up a lot more opportunities for optimization, so Emacs Lisp code that makes use of lexical binding is likely to run faster in future Emacs versions. Such code is also much more friendly to concurrency, which we want to add to Emacs in the near future.

11.9.4 Using Lexical Binding

When loading an Emacs Lisp file or evaluating a Lisp buffer, lexical binding is enabled if the buffer-local variable `lexical-binding` is non-`nil`:

`lexical-binding` [Variable]

> If this buffer-local variable is non-`nil`, Emacs Lisp files and buffers are evaluated using lexical binding instead of dynamic binding. (However, special variables are still dynamically bound; see below.) If `nil`, dynamic binding is used for all local variables. This variable is typically set for a whole Emacs Lisp file, as a file local variable (see Section 11.11 [File Local Variables], page 156). Note that unlike other such variables, this one must be set in the first line of a file.

When evaluating Emacs Lisp code directly using an `eval` call, lexical binding is enabled if the *lexical* argument to `eval` is non-`nil`. See Section 9.4 [Eval], page 117.

Even when lexical binding is enabled, certain variables will continue to be dynamically bound. These are called *special variables*. Every variable that has been defined with `defvar`, `defcustom` or `defconst` is a special variable (see Section 11.5 [Defining Variables], page 141). All other variables are subject to lexical binding.

`special-variable-p` *SYMBOL* [Function]

> This function returns non-`nil` if *symbol* is a special variable (i.e. it has a `defvar`, `defcustom`, or `defconst` variable definition). Otherwise, the return value is `nil`.

The use of a special variable as a formal argument in a function is discouraged. Doing so gives rise to unspecified behavior when lexical binding mode is enabled (it may use lexical binding sometimes, and dynamic binding other times).

Converting an Emacs Lisp program to lexical binding is pretty easy. First, add a file-local variable setting of `lexical-binding` to `t` in the Emacs Lisp source file. Second, check that every variable in the program which needs to be dynamically bound has a variable definition, so that it is not inadvertently bound lexically.

A simple way to find out which variables need a variable definition is to byte-compile the source file. See Chapter 16 [Byte Compilation], page 223. If a non-special variable is used outside of a `let` form, the byte-compiler will warn about reference or assignment to a “free variable”. If a non-special variable is bound but not used within a `let` form, the byte-compiler will warn about an “unused lexical variable”. The byte-compiler will also issue a warning if you use a special variable as a function argument.

(To silence byte-compiler warnings about unused variables, just use a variable name that start with an underscore. The byte-compiler interprets this as an indication that this is a variable known not to be used.)

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, such as *buffer-local* bindings, which apply only in one buffer. Having different values for a variable in different buffers is an important customization method. (Variables can also have bindings that are local to each terminal. See Section 29.2 [Multiple Terminals], page 67, vol. 2.)

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, 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 399.

The usual way to make a buffer-local binding is with `make-local-variable`, which is what major mode commands typically use. This affects just the current buffer; all other buffers (including those yet to be created) will continue to share the default value unless they are explicitly given their own buffer-local bindings.

A more powerful operation is to mark the variable as *automatically buffer-local* by calling `make-variable-buffer-local`. You can think of this as making the variable local in all buffers, even those yet to be created. More precisely, the effect is that setting the variable automatically makes the variable local to the current buffer if it is not already so. All buffers start out by sharing the default value of the variable as usual, but setting the variable creates a buffer-local binding for the current buffer. The new value is stored in the buffer-local binding, leaving the default binding untouched. This means that the default value cannot be changed with `setq` in any buffer; the only way to change it is with `setq-default`.

Warning: When a variable has buffer-local bindings in one or more buffers, `let` rebinds the binding that's currently in effect. For instance, if the current buffer has a buffer-local value, `let` temporarily rebinds that. If no buffer-local bindings are in effect, `let` rebinds the default value. If inside the `let` you then change to a different current buffer in which a different binding is in effect, you won't see the `let` binding any more. And if you exit the `let` while still in the other buffer, you won't see the unbinding occur (though it will occur properly). Here is an example to illustrate:

```
(setq foo 'g)
(set-buffer "a")
(make-local-variable 'foo)
(setq foo 'a)
(let ((foo 'temp))
  ;; foo ⇒ 'temp  ; let binding in buffer 'a'
  (set-buffer "b")
  ;; foo ⇒ 'g     ; the global value since foo is not local in 'b'
  body...)
```

```
foo ⇒ 'g           ; exiting restored the local value in buffer ‘a’,
                   ; but we don’t see that in buffer ‘b’
(set-buffer "a")   ; verify the local value was restored
foo ⇒ 'a
```

Note that references to `foo` in *body* access the buffer-local binding of buffer ‘`b`’.

When a file specifies local variable values, these become buffer-local values when you visit the file. See Section “File Variables” in *The GNU Emacs Manual*.

A buffer-local variable cannot be made terminal-local (see Section 29.2 [Multiple Terminals], page 67, vol. 2).

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.
(with-current-buffer "b2"
  foo)
     ⇒ 5
```

Making a variable buffer-local within a `let`-binding for that variable does not work reliably, unless the buffer in which you do this is not current either on entry to or exit from the `let`. This is because `let` does not distinguish between different kinds of bindings; it knows only which variable the binding was made for.

If the variable is terminal-local (see Section 29.2 [Multiple Terminals], page 67, vol. 2), this function signals an error. Such variables cannot have buffer-local bindings as well.

Warning: do not use `make-local-variable` for a hook variable. The hook variables are automatically made buffer-local as needed if you use the *local* argument to `add-hook` or `remove-hook`.

`make-variable-buffer-local` *variable* [Command]

This function marks *variable* (a symbol) automatically buffer-local, so that any subsequent attempt to set it will make it local to the current buffer at the time. Unlike `make-local-variable`, with which it is often confused, this cannot be undone, and affects the behavior of the variable in all buffers.

A peculiar wrinkle of this feature is that binding the variable (with `let` or other binding constructs) does not create a buffer-local binding for it. Only setting the variable (with `set` or `setq`), while the variable does not have a `let`-style binding that was made in the current buffer, does so.

If *variable* does not have a default value, then calling this command will give it a default value of `nil`. If *variable* already has a default value, that value remains unchanged. Subsequently calling `makunbound` on *variable* will result in a void buffer-local value and leave the default value unaffected.

The value returned is *variable*.

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 155) 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.) Normally, each list element has the form `(sym . val)`, where *sym* is a buffer-local variable (a symbol) and *val* is its buffer-local value. But when a variable's buffer-local binding in *buffer* is void, its list element is just *sym*.

```
(make-local-variable 'foobar)
(makunbound 'foobar)
(make-local-variable 'bind-me)
(setq bind-me 69)
(setq lcl (buffer-local-variables))
    ;; First, built-in variables local in all buffers:
⇒ ((mark-active . nil)
    (buffer-undo-list . nil)
```

```
          (mode-name . "Fundamental")
          ...
          ;; Next, non-built-in buffer-local variables.
          ;; This one is buffer-local and void:
          foobar
          ;; This one is buffer-local and nonvoid:
          (bind-me . 69))
```

Note that storing new values into the CDRs of cons cells in this list does *not* change the buffer-local values of the variables.

`kill-local-variable` *variable* [Command]

This function deletes the buffer-local binding (if any) for *variable* (a symbol) in the current buffer. As a result, the default binding of *variable* becomes visible in this buffer. This typically results in a change in the value of *variable*, since the default value is usually different from the buffer-local value just eliminated.

If you kill the buffer-local binding of a variable that automatically becomes buffer-local when set, this makes the default value visible in the current buffer. However, if you set the variable again, that will once again create a buffer-local binding for it.

`kill-local-variable` returns *variable*.

This function is a command because it is sometimes useful to kill one buffer-local variable interactively, just as it is useful to create buffer-local variables interactively.

`kill-all-local-variables` [Function]

This function eliminates all the buffer-local variable bindings of the current buffer except for variables marked as "permanent" and local hook functions that have a non-`nil` `permanent-local-hook` property (see Section 23.1.2 [Setting Hooks], page 398). 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 396.

A buffer-local variable is *permanent* if the variable name (a symbol) has a `permanent-local` property that is non-`nil`. Such variables are unaffected by `kill-all-local-variables`, and their local bindings are therefore not cleared by changing major modes. Permanent locals are appropriate for data pertaining to where the file came from or how to save it, rather than with how to edit the contents.

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 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 144).

`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 File Local Variables

A file can specify local variable values; Emacs uses these to create buffer-local bindings for those variables in the buffer visiting that file. See Section "Local Variables in Files" in *The GNU Emacs Manual*, for basic information about file-local variables. This section describes the functions and variables that affect how file-local variables are processed.

If a file-local variable could specify an arbitrary function or Lisp expression that would be called later, visiting a file could take over your Emacs. Emacs protects against this by automatically setting only those file-local variables whose specified values are known to be safe. Other file-local variables are set only if the user agrees.

For additional safety, `read-circle` is temporarily bound to `nil` when Emacs reads file-local variables (see Section 19.3 [Input Functions], page 276). This prevents the Lisp reader from recognizing circular and shared Lisp structures (see Section 2.5 [Circular Objects], page 26).

`enable-local-variables` [User Option]

This variable controls whether to process file-local variables. The possible values are:

`t` (the default)
: Set the safe variables, and query (once) about any unsafe variables.

`:safe`
: Set only the safe variables and do not query.

`:all` Set all the variables and do not query.

`nil` Don't set any variables.

anything else
Query (once) about all the variables.

`inhibit-local-variables-regexps` [Variable]
This is a list of regular expressions. If a file has a name matching an element of this list, then it is not scanned for any form of file-local variable. For examples of why you might want to use this, see Section 23.2.2 [Auto Major Mode], page 403.

`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 23.2.2 [Auto Major Mode], page 403).

This function works by walking the alist stored in `file-local-variables-alist` and applying each local variable in turn. It calls `before-hack-local-variables-hook` and `hack-local-variables-hook` before and after applying the variables, respectively. It only calls the before-hook if the alist is non-`nil`; it always calls the other hook. This function ignores a '`mode`' element if it specifies the same major mode as the buffer already has.

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

`file-local-variables-alist` [Variable]
This buffer-local variable holds the alist of file-local variable settings. Each element of the alist is of the form `(var . value)`, where *var* is a symbol of the local variable and *value* is its value. When Emacs visits a file, it first collects all the file-local variables into this alist, and then the `hack-local-variables` function applies them one by one.

`before-hack-local-variables-hook` [Variable]
Emacs calls this hook immediately before applying file-local variables stored in `file-local-variables-alist`.

`hack-local-variables-hook` [Variable]
Emacs calls this hook immediately after it finishes applying file-local variables stored in `file-local-variables-alist`.

You can specify safe values for a variable with a `safe-local-variable` property. The property has to be a function of one argument; any value is safe if the function returns non-`nil` given that value. Many commonly-encountered file variables have `safe-local-variable` properties; these include `fill-column`, `fill-prefix`, and `indent-tabs-mode`. For boolean-valued variables that are safe, use `booleanp` as the property value. Lambda expressions should be quoted so that `describe-variable` can display the predicate.

When defining a user option using `defcustom`, you can set its `safe-local-variable` property by adding the arguments `:safe` *`function`* to `defcustom` (see Section 14.3 [Variable Definitions], page 193).

`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*. If a variable is risky, it is never entered automatically into `safe-local-variable-values`; Emacs always queries before setting a risky variable, unless the user explicitly allows a value by customizing `safe-local-variable-values` directly.

Any variable whose name has a non-`nil` `risky-local-variable` property is considered risky. When you define a user option using `defcustom`, you can set its `risky-local-variable` property by adding the arguments `:risky` *`value`* to `defcustom` (see Section 14.3 [Variable Definitions], page 193). In addition, any variable whose name ends in any of '`-command`', '`-frame-alist`', '`-function`', '`-functions`', '`-hook`', '`-hooks`', '`-form`', '`-forms`', '`-map`', '`-map-alist`', '`-mode-alist`', '`-program`', or '`-predicate`' is automatically considered risky. The variables '`font-lock-keywords`', '`font-lock-keywords`' followed by a digit, and '`font-lock-syntactic-keywords`' are also considered risky.

`risky-local-variable-p` *sym* [Function]

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

`ignored-local-variables` [Variable]

> This variable holds a list of variables that should not be given local values by files. Any value specified for one of these variables is completely ignored.

The '`Eval:`' "variable" is also a potential loophole, so Emacs normally asks for confirmation before handling it.

`enable-local-eval` [User Option]

> This variable controls processing of '`Eval:`' in '`-*-`' lines or local variables lists in files being visited. A value of `t` means process them unconditionally; `nil` means ignore them; anything else means ask the user what to do for each file. The default value is `maybe`.

`safe-local-eval-forms` [User Option]

> This variable holds a list of expressions that are safe to evaluate when found in the '`Eval:`' "variable" in a file local variables list.

If the expression is a function call and the function has a `safe-local-eval-function` property, the property value determines whether the expression is safe to evaluate. The property value can be a predicate to call to test the expression, a list of such predicates (it's safe if any predicate succeeds), or `t` (always safe provided the arguments are constant).

Text properties are also potential loopholes, since their values could include functions to call. So Emacs discards all text properties from string values specified for file-local variables.

11.12 Directory Local Variables

A directory can specify local variable values common to all files in that directory; Emacs uses these to create buffer-local bindings for those variables in buffers visiting any file in that directory. This is useful when the files in the directory belong to some *project* and therefore share the same local variables.

There are two different methods for specifying directory local variables: by putting them in a special file, or by defining a *project class* for that directory.

`dir-locals-file` [Constant]

This constant is the name of the file where Emacs expects to find the directory-local variables. The name of the file is '`.dir-locals.el`'[2]. A file by that name in a directory causes Emacs to apply its settings to any file in that directory or any of its subdirectories (optionally, you can exclude subdirectories; see below). If some of the subdirectories have their own '`.dir-locals.el`' files, Emacs uses the settings from the deepest file it finds starting from the file's directory and moving up the directory tree. The file specifies local variables as a specially formatted list; see Section "Per-directory Local Variables" in *The GNU Emacs Manual*, for more details.

`hack-dir-local-variables` [Function]

This function reads the `.dir-locals.el` file and stores the directory-local variables in `file-local-variables-alist` that is local to the buffer visiting any file in the directory, without applying them. It also stores the directory-local settings in `dir-locals-class-alist`, where it defines a special class for the directory in which '`.dir-locals.el`' file was found. This function works by calling `dir-locals-set-class-variables` and `dir-locals-set-directory-class`, described below.

`hack-dir-local-variables-non-file-buffer` [Function]

This function looks for directory-local variables, and immediately applies them in the current buffer. It is intended to be called in the mode commands for non-file buffers, such as Dired buffers, to let them obey directory-local variable settings. For non-file buffers, Emacs looks for directory-local variables in `default-directory` and its parent directories.

`dir-locals-set-class-variables` *class variables* [Function]

This function defines a set of variable settings for the named *class*, which is a symbol. You can later assign the class to one or more directories, and Emacs will apply those variable settings to all files in those directories. The list in *variables* can be of one of the two forms: `(`*`major-mode . alist`*`)` or `(`*`directory . list`*`)`. With the first form,

[2] The MS-DOS version of Emacs uses '`_dir-locals.el`' instead, due to limitations of the DOS filesystems.

if the file's buffer turns on a mode that is derived from *major-mode*, then the all the variables in the associated *alist* are applied; *alist* should be of the form `(name . value)`. A special value `nil` for *major-mode* means the settings are applicable to any mode. In *alist*, you can use a special *name*: `subdirs`. If the associated value is `nil`, the alist is only applied to files in the relevant directory, not to those in any subdirectories.

With the second form of *variables*, if *directory* is the initial substring of the file's directory, then *list* is applied recursively by following the above rules; *list* should be of one of the two forms accepted by this function in *variables*.

`dir-locals-set-directory-class` *directory class* **&optional** *mtime* [Function]

This function assigns *class* to all the files in `directory` and its subdirectories. Thereafter, all the variable settings specified for *class* will be applied to any visited file in *directory* and its children. *class* must have been already defined by `dir-locals-set-class-variables`.

Emacs uses this function internally when it loads directory variables from a `.dir-locals.el` file. In that case, the optional argument *mtime* holds the file modification time (as returned by `file-attributes`). Emacs uses this time to check stored local variables are still valid. If you are assigning a class directly, not via a file, this argument should be `nil`.

`dir-locals-class-alist` [Variable]

This alist holds the class symbols and the associated variable settings. It is updated by `dir-locals-set-class-variables`.

`dir-locals-directory-cache` [Variable]

This alist holds directory names, their assigned class names, and modification times of the associated directory local variables file (if there is one). The function `dir-locals-set-directory-class` updates this list.

11.13 Variable Aliases

It is sometimes useful to make two variables synonyms, so that both variables always have the same value, and changing either one also changes the other. Whenever you change the name of a variable—either because you realize its old name was not well chosen, or because its meaning has partly changed—it can be useful to keep the old name as an *alias* of the new one for compatibility. You can do this with `defvaralias`.

`defvaralias` *new-alias base-variable* **&optional** *docstring* [Function]

This function defines the symbol *new-alias* as a variable alias for symbol *base-variable*. This means that retrieving the value of *new-alias* returns the value of *base-variable*, and changing the value of *new-alias* changes the value of *base-variable*. The two aliased variable names always share the same value and the same bindings.

If the *docstring* argument is non-`nil`, it specifies the documentation for *new-alias*; otherwise, the alias gets the same documentation as *base-variable* has, if any, unless *base-variable* is itself an alias, in which case *new-alias* gets the documentation of the variable at the end of the chain of aliases.

This function returns *base-variable*.

Variable aliases are convenient for replacing an old name for a variable with a new name. `make-obsolete-variable` declares that the old name is obsolete and therefore that it may be removed at some stage in the future.

`make-obsolete-variable` *obsolete-name current-name when* **&optional** *access-type* [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. *when* should be a string indicating when the variable was first made obsolete (usually a version number string).

The optional argument *access-type*, if non-`nil`, should should specify the kind of access that will trigger obsolescence warnings; it can be either `get` or `set`.

You can make two variables synonyms and declare one obsolete at the same time using the macro `define-obsolete-variable-alias`.

`define-obsolete-variable-alias` *obsolete-name current-name* **&optional** *when docstring* [Macro]

This macro marks the variable *obsolete-name* as obsolete and also makes it an alias for the variable *current-name*. It is equivalent to the following:

```
(defvaralias obsolete-name current-name docstring)
(make-obsolete-variable obsolete-name current-name when)
```

`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
```

11.14 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 465, vol. 2, 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 undo-limit 1000.0)
error  Wrong type argument: integerp, 1000.0
```

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 out a computation given input values called *arguments*. The result of the computation is called the *value* or *return value* of the function. The computation can also have side effects, such as lasting changes in the values of variables or the contents of data structures.

In most computer languages, every function has a name. But in Lisp, a function in the strictest sense has no name: it is an object which can *optionally* be associated with a symbol (e.g. `car`) that serves as the function name. See Section 12.3 [Function Names], page 168. When a function has been given a name, we usually also refer to that symbol as a “function” (e.g. we refer to “the function `car`”). In this manual, the distinction between a function name and the function object itself is usually unimportant, but we will take note wherever it is relevant.

Certain function-like objects, called *special forms* and *macros*, also accept arguments to carry out computations. However, as explained below, these are not considered functions in Emacs Lisp.

Here are important terms for functions and function-like objects:

lambda expression
: A function (in the strict sense, i.e. a function object) which is written in Lisp. These are described in the following section.

primitive
: A function which is callable from Lisp but is actually written in C. Primitives are also called *built-in functions*, or *subrs*. Examples include functions like `car` and `append`. In addition, all special forms (see below) are also considered primitives.

  Usually, a function is implemented as a primitive because it is a fundamental part of Lisp (e.g. `car`), or because it provides a low-level interface to operating system services, or because it needs to run fast. Unlike functions defined in Lisp, primitives can be modified or added only by changing the C sources and recompiling Emacs. See Section E.5 [Writing Emacs Primitives], page 463, vol. 2.

special form
: A primitive that is like a function but does not evaluate all of its arguments in the usual way. It may evaluate only some of the arguments, or may evaluate them in an unusual order, or several times. Examples include `if`, `and`, and `while`. See Section 9.1.7 [Special Forms], page 114.

macro
: A construct defined in Lisp, which differs from a function in that it translates a Lisp expression into another expression which is to be evaluated instead of the original expression. Macros enable Lisp programmers to do the sorts of things that special forms can do. See Chapter 13 [Macros], page 181.

command An object which can be invoked via the `command-execute` primitive, usually due to the user typing in a key sequence *bound* to that command. See Section 21.3 [Interactive Call], page 321. A command is usually a function; if the function is written in Lisp, it is made into a command by an `interactive` form in the function definition (see Section 21.2 [Defining Commands], page 316). Commands that are functions can also be called from Lisp expressions, just like other functions.

Keyboard macros (strings and vectors) are commands also, even though they are not functions. See Section 21.16 [Keyboard Macros], page 358. We say that a symbol is a command if its function cell contains a command (see Section 8.1 [Symbol Components], page 102); such a *named command* can be invoked with `M-x`.

closure A function object that is much like a lambda expression, except that it also encloses an "environment" of lexical variable bindings. See Section 12.9 [Closures], page 176.

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

autoload object
A place-holder for a real function. If the autoload object is called, Emacs loads the file containing the definition of the real function, and then calls the real function. See Section 15.5 [Autoload], page 213.

You can use the function `functionp` to test if an object is a function:

`functionp` *object* [Function]

This function returns `t` if *object* is any kind of function, i.e. can be passed to `funcall`. Note that `functionp` returns `t` for symbols that are function names, and returns `nil` for special forms.

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 lambda expression is a function object written in Lisp. Here is an example:

```
(lambda (x)
  "Return the hyperbolic cosine of X."
  (* 0.5 (+ (exp x) (exp (- x)))))
```

In Emacs Lisp, such a list is valid as an expression—it evaluates to itself. But its main use is not to be evaluated as an expression, but to be called as a function.

A lambda expression, by itself, has no name; it is an *anonymous function*. Although lambda expressions can be used this way (see Section 12.7 [Anonymous Functions], page 174), they are more commonly associated with symbols to make *named functions* (see Section 12.3 [Function Names], page 168). Before going into these details, the following subsections describe the components of a lambda expression and what they do.

12.2.1 Components of a Lambda Expression

A lambda expression is a list that looks like this:

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

The first element of a lambda expression is always the symbol `lambda`. This indicates that the list represents a function. The reason functions are defined to start with `lambda` is so that other lists, intended for other uses, will not accidentally be valid as functions.

The second element is a list of symbols—the argument variable names. 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 138.

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 167.

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 316, 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 the following example:

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

We can call this function by passing it to `funcall`, like this:

```
(funcall (lambda (a b c) (+ a b c))
         1 2 3)
```

This call evaluates the body of the lambda expression with the variable `a` bound to 1, `b` bound to 2, and `c` bound to 3. Evaluation of the body adds these three numbers, producing the result 6; therefore, this call to the function returns the value 6.

Note that the arguments can be the results of other function calls, as in this example:

```
(funcall (lambda (a b c) (+ a b c))
         1 (* 2 3) (- 5 4))
```

This evaluates the arguments `1`, `(* 2 3)`, and `(- 5 4)` from left to right. Then it applies the lambda expression to the argument values 1, 6 and 1 to produce the value 8.

As these examples show, you can use a form with a lambda expression as its CAR to make local variables and give them values. In the old days of Lisp, this technique was the only way to bind and initialize local variables. But nowadays, it is clearer to use the special form `let` for this purpose (see Section 11.3 [Local Variables], page 138). Lambda expressions are mainly used as anonymous functions for passing as arguments to other functions (see Section 12.7 [Anonymous Functions], page 174), or stored as symbol function definitions to produce named functions (see Section 12.3 [Function Names], page 168).

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:

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

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 451, 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

A symbol can serve as the name of a function. This happens when the symbol's *function cell* (see Section 8.1 [Symbol Components], page 102) contains a function object (e.g. a lambda expression). Then the symbol itself becomes a valid, callable function, equivalent to the function object in its function cell.

The contents of the function cell are also called the symbol's *function definition*. The procedure of using a symbol's function definition in place of the symbol is called *symbol function indirection*; see Section 9.1.4 [Function Indirection], page 112. If you have not given a symbol a function definition, its function cell is said to be *void*, and it cannot be used as a function.

In practice, nearly all functions have names, and are referred to by their names. You can create a named Lisp function by defining a lambda expression and putting it in a function cell (see Section 12.8 [Function Cells], page 175). However, it is more common to use the `defun` special form, described in the next section.

We give functions names because it is convenient to refer to them by their names in Lisp expressions. Also, a named Lisp function can easily refer to itself—it can be recursive. Furthermore, primitives can only be referred to textually by their names, since primitive function objects (see Section 2.3.15 [Primitive Function Type], page 22) have no read syntax.

A function need not have a unique name. A given function object *usually* appears in the function cell of only one symbol, but this is just a convention. It is easy to store it in several symbols using `fset`; then each of the symbols is a valid name for the same function.

Note that a symbol used as a function name may also be used as a variable; these two uses of a symbol are independent and do not conflict. (This is not the case in some dialects of Lisp, like Scheme.)

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`. Also, the first two of the *body-forms* may be a documentation string and an interactive declaration. See Section 12.2.1 [Lambda Components], page 165.

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 ()
  "Upcase the last letter of the word at point."
  (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. Emacs does not prevent you from doing this, because redefining a function is sometimes done deliberately, and there is no way to distinguish deliberate redefinition from unintentional redefinition.

`defalias` *name definition* **&optional** *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 220).

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 175.

You cannot create a new primitive function with `defun` or `defalias`, but you can use them to change the function definition of any symbol, even one such as `car` or `x-popup-menu` whose normal definition is a primitive. However, this is risky: for instance, it is next to impossible to redefine `car` without breaking Lisp completely. Redefining an obscure function such as `x-popup-menu` is less dangerous, but it still may not work as you expect. If there are calls to the primitive from C code, they call the primitive's C definition directly, so changing the symbol's definition will have no effect on them.

See also `defsubst`, which defines a function like `defun` and tells the Lisp compiler to perform inline expansion on it. See Section 12.11 [Inline Functions], page 177.

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 172.

Sometimes it is useful to fix some of the function's arguments at certain values, and leave the rest of arguments for when the function is actually called. The act of fixing some of the function's arguments is called *partial application* of the function[1]. The result is a new function that accepts the rest of arguments and calls the original function with all the arguments combined.

Here's how to do partial application in Emacs Lisp:

`apply-partially` *func* **&rest** *args* [Function]

This function returns a new function which, when called, will call *func* with the list of arguments composed from *args* and additional arguments specified at the time of

[1] This is related to, but different from *currying*, which transforms a function that takes multiple arguments in such a way that it can be called as a chain of functions, each one with a single argument.

the call. If *func* accepts *n* arguments, then a call to `apply-partially` with $m < n$ arguments will produce a new function of $n - m$ arguments.

Here's how we could define the built-in function `1+`, if it didn't exist, using `apply-partially` and `+`, another built-in function:

```
(defalias '1+ (apply-partially '+ 1)
  "Increment argument by one.")
(1+ 10)
     ⇒ 11
```

It is common for Lisp functions to accept functions as arguments or find them in data structures (especially in hook variables and property lists) and call them using `funcall` or `apply`. Functions that accept function arguments are often called *functionals*.

Sometimes, when you call a functional, it is useful to supply a no-op function as the argument. Here are 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`.

Some functions are user-visible *commands*, which can be called interactively (usually by a key sequence). It is possible to invoke such a command exactly as though it was called interactively, by using the `call-interactively` function. See Section 21.3 [Interactive Call], page 321.

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; this section describes `mapcar`, `mapc`, and `mapconcat`, which map over a list. See [Definition of mapatoms], page 106, for the function `mapatoms` which maps over the symbols in an obarray. See [Definition of maphash], page 100, 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 92).

`mapcar` *function sequence* [Function]

`mapcar` applies *function* to each element of *sequence* in turn, and returns a list of the results.

The argument *sequence* can be any kind of sequence except a char-table; that is, a list, a vector, a bool-vector, or a string. The result is always a list. The length of the result is the same as the length of *sequence*. For example:

```
(mapcar 'car '((a b) (c d) (e f)))
     ⇒ (a c e)
(mapcar '1+ [1 2 3])
     ⇒ (2 3 4)
(mapcar 'string "abc")
     ⇒ ("a" "b" "c")

;; Call each function in my-hooks.
(mapcar 'funcall my-hooks)

(defun mapcar* (function &rest args)
  "Apply FUNCTION to successive cars of all ARGS.
Return the list of results."
  ;; If no list is exhausted,
  (if (not (memq nil args))
      ;; apply function to CARs.
      (cons (apply function (mapcar 'car args))
            (apply 'mapcar* function
                   ;; Recurse for rest of elements.
                   (mapcar 'cdr args)))))

(mapcar* 'cons '(a b c) '(1 2 3 4))
     ⇒ ((a . 1) (b . 2) (c . 3))
```

`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

Although functions are usually defined with `defun` and given names at the same time, it is sometimes convenient to use an explicit lambda expression—an *anonymous function*. Anonymous functions are valid wherever function names are. They are often assigned as variable values, or as arguments to functions; for instance, you might pass one as the *function* argument to `mapcar`, which applies that function to each element of a list (see Section 12.6 [Mapping Functions], page 172). See [describe-symbols example], page 453, for a realistic example of this.

When defining a lambda expression that is to be used as an anonymous function, you can in principle use any method to construct the list. But typically you should use the `lambda` macro, or the `function` special form, or the `#'` read syntax:

`lambda` *args body...* [Macro]

This macro returns an anonymous function with argument list *args* and body forms given by *body*. In effect, this macro makes `lambda` forms "self-quoting": evaluating a form whose CAR is `lambda` yields the form itself:

```
(lambda (x) (* x x))
     ⇒ (lambda (x) (* x x))
```

The `lambda` form has one other effect: it tells the Emacs evaluator and byte-compiler that its argument is a function, by using `function` as a subroutine (see below).

`function` *function-object* [Special Form]

This special form returns *function-object* without evaluating it. In this, it is similar to `quote` (see Section 9.2 [Quoting], page 116). But unlike `quote`, it also serves as a note to the Emacs evaluator and byte-compiler that *function-object* is intended to be used as a function. Assuming *function-object* is a valid lambda expression, this has two effects:

- When the code is byte-compiled, *function-object* is compiled into a byte-code function object (see Chapter 16 [Byte Compilation], page 223).
- When lexical binding is enabled, *function-object* is converted into a closure. See Section 12.9 [Closures], page 176.

The read syntax `#'` is a short-hand for using `function`. The following forms are all equivalent:

```
(lambda (x) (* x x))
(function (lambda (x) (* x x)))
#'(lambda (x) (* x x))
```

In the following example, we define a `change-property` function that takes a function as its third argument, followed by a `double-property` function that makes use of `change-property` by passing it an anonymous function:

```
(defun change-property (symbol prop function)
  (let ((value (get symbol prop)))
    (put symbol prop (funcall function value))))

(defun double-property (symbol prop)
  (change-property symbol prop (lambda (x) (* 2 x))))
```

Note that we do not quote the `lambda` form.

If you compile the above code, the anonymous function is also compiled. This would not happen if, say, you had constructed the anonymous function by quoting it as a list:

```
(defun double-property (symbol prop)
  (change-property symbol prop '(lambda (x) (* 2 x))))
```

In that case, the anonymous function is kept as a lambda expression in the compiled code. The byte-compiler cannot assume this list is a function, even though it looks like one, since it does not know that `change-property` intends to use it as a function.

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 140.)

```
(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.

The primary use of this function is as a subroutine by constructs that define or alter functions, like `defadvice` (see Chapter 17 [Advising Functions], page 233). (If `defun` were not a primitive, it could be written as a Lisp macro using `fset`.) You can also use it to give a symbol a function definition that is not a list, e.g. a keyboard macro (see Section 21.16 [Keyboard Macros], page 358):

```
;; Define a named keyboard macro.
(fset 'kill-two-lines "\^u2\^k")
     ⇒ "\^u2\^k"
```

It you wish to use `fset` to make an alternate name for a function, consider using `defalias` instead. See [Definition of defalias], page 170.

12.9 Closures

As explained in Section 11.9 [Variable Scoping], page 146, Emacs can optionally enable lexical binding of variables. When lexical binding is enabled, any named function that you create (e.g. with `defun`), as well as any anonymous function that you create using the `lambda` macro or the `function` special form or the `#'` syntax (see Section 12.7 [Anonymous Functions], page 174), is automatically converted into a closure.

A closure is a function that also carries a record of the lexical environment that existed when the function was defined. When it is invoked, any lexical variable references within its definition use the retained lexical environment. In all other respects, closures behave much like ordinary functions; in particular, they can be called in the same way as ordinary functions.

See Section 11.9.3 [Lexical Binding], page 148, for an example of using a closure.

Currently, an Emacs Lisp closure object is represented by a list with the symbol `closure` as the first element, a list representing the lexical environment as the second element, and the argument list and body forms as the remaining elements:

```
;; lexical binding is enabled.
(lambda (x) (* x x))
     ⇒ (closure (t) (x) (* x x))
```

However, the fact that the internal structure of a closure is "exposed" to the rest of the Lisp world is considered an internal implementation detail. For this reason, we recommend against directly examining or altering the structure of closure objects.

12.10 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)
```

In addition, you can mark a certain a particular calling convention for a function as obsolete:

`set-advertised-calling-convention` *function signature when* [Function]

This function specifies the argument list *signature* as the correct way to call *function*. This causes the Emacs byte compiler to issue a warning whenever it comes across an Emacs Lisp program that calls *function* any other way (however, it will still allow the code to be byte compiled). *when* should be a string indicating when the variable was first made obsolete (usually a version number string).

For instance, in old versions of Emacs the `sit-for` function accepted three arguments, like this

```
  (sit-for seconds milliseconds nodisp)
```

However, calling `sit-for` this way is considered obsolete (see Section 21.10 [Waiting], page 350). The old calling convention is deprecated like this:

```
(set-advertised-calling-convention
  'sit-for '(seconds &optional nodisp) "22.1")
```

12.11 Inline Functions

`defsubst` *name argument-list body-forms...* [Macro]

Define an inline function. The syntax is exactly the same as `defun` (see Section 12.4 [Defining Functions], page 169).

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 byte-compile a call to the function (see Chapter 16 [Byte Compilation], page 223), the function's definition is expanded into the caller.

Making a function inline often makes its function calls run faster. But it also has disadvantages. For one thing, it reduces flexibility; if you change the definition of the function, calls already inlined still use the old definition until you recompile them.

Another disadvantage is that making a large function inline can increase the size of compiled code both in files and in memory. Since the speed advantage of inline functions is greatest for small functions, you generally should not make large functions inline.

Also, inline functions do not behave well with respect to debugging, tracing, and advising (see Chapter 17 [Advising Functions], page 233). 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 181). But the macro would be limited to direct use in expressions—a macro cannot be called with `apply`, `mapcar` and so on. Also, it takes some work to convert an ordinary function into a macro. To convert it into an inline function is easy; just replace `defun` with `defsubst`. Since each argument of an inline function is evaluated exactly once, you needn't worry about how many times the body uses the arguments, as you do for macros.

After an inline function is defined, its inline expansion can be performed later on in the same file, just like macros.

12.12 Telling the Compiler that a Function is Defined

Byte-compiling a file often produces warnings about functions that the compiler doesn't know about (see Section 16.6 [Compiler Errors], page 228). Sometimes this indicates a real problem, but usually the functions in question are defined in other files which would be loaded if that code is run. For example, byte-compiling '`fortran.el`' used to warn:

```
In end of data:
fortran.el:2152:1:Warning: the function ‘gud-find-c-expr’ is not
    known to be defined.
```

In fact, `gud-find-c-expr` is only used in the function that Fortran mode uses for the local value of `gud-find-expr-function`, which is a callback from GUD; if it is called, the GUD functions will be loaded. When you know that such a warning does not indicate a real problem, it is good to suppress the warning. That makes new warnings which might mean real problems more visible. You do that with `declare-function`.

All you need to do is add a `declare-function` statement before the first use of the function in question:

```
(declare-function gud-find-c-expr "gud.el" nil)
```

This says that `gud-find-c-expr` is defined in '`gud.el`' (the '`.el`' can be omitted). The compiler takes for granted that that file really defines the function, and does not check.

The optional third argument specifies the argument list of `gud-find-c-expr`. In this case, it takes no arguments (`nil` is different from not specifying a value). In other cases, this might be something like `(file &optional overwrite)`. You don't have to specify the argument list, but if you do the byte compiler can check that the calls match the declaration.

`declare-function` *function file* **&optional** *arglist fileonly* [Macro]

> Tell the byte compiler to assume that *function* is defined, with arguments *arglist*, and that the definition should come from the file *file*. *fileonly* non-`nil` means only check that *file* exists, not that it actually defines *function*.

To verify that these functions really are declared where `declare-function` says they are, use `check-declare-file` to check all `declare-function` calls in one source file, or use `check-declare-directory` check all the files in and under a certain directory.

These commands find the file that ought to contain a function's definition using `locate-library`; if that finds no file, they expand the definition file name relative to the directory of the file that contains the `declare-function` call.

You can also say that a function is a primitive by specifying a file name ending in '`.c`' or '`.m`'. This is useful only when you call a primitive that is defined only on certain systems. Most primitives are always defined, so they will never give you a warning.

Sometimes a file will optionally use functions from an external package. If you prefix the filename in the `declare-function` statement with '`ext:`', then it will be checked if it is found, otherwise skipped without error.

There are some function definitions that '`check-declare`' does not understand (e.g. `defstruct` and some other macros). In such cases, you can pass a non-`nil` *fileonly* argument to `declare-function`, meaning to only check that the file exists, not that it actually defines the function. Note that to do this without having to specify an argument list, you should set the *arglist* argument to `t` (because `nil` means an empty argument list, as opposed to an unspecified one).

12.13 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.

12.14 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 170.
`autoload`	See Section 15.5 [Autoload], page 213.
`call-interactively`	See Section 21.3 [Interactive Call], page 321.
`called-interactively-p`	See Section 21.4 [Distinguish Interactive], page 323.
`commandp`	See Section 21.3 [Interactive Call], page 321.
`documentation`	See Section 24.2 [Accessing Documentation], page 452.
`eval`	See Section 9.4 [Eval], page 117.
`funcall`	See Section 12.5 [Calling Functions], page 170.
`function`	See Section 12.7 [Anonymous Functions], page 174.
`ignore`	See Section 12.5 [Calling Functions], page 170.
`indirect-function`	See Section 9.1.4 [Function Indirection], page 112.
`interactive`	See Section 21.2.1 [Using Interactive], page 316.
`interactive-p`	See Section 21.4 [Distinguish Interactive], page 323.
`mapatoms`	See Section 8.3 [Creating Symbols], page 104.
`mapcar`	See Section 12.6 [Mapping Functions], page 172.
`map-char-table`	See Section 6.6 [Char-Tables], page 92.
`mapconcat`	See Section 12.6 [Mapping Functions], page 172.
`undefined`	See Section 22.11 [Functions for Key Lookup], page 373.

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 12.11 [Inline Functions], page 177.

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 an alternate Lisp expression, also known as the *expansion* of the macro. The Lisp interpreter proceeds to evaluate the expansion as soon as it comes back from the macro.

Since the expansion is evaluated in the normal manner, it may contain calls to other macros. It may even be a call to the same macro, though this is unusual.

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.

Byte-compiling a file also executes any `require` calls at top-level in the file, so you can ensure that necessary macro definitions are available during compilation by requiring the files that define them (see Section 15.7 [Named Features], page 217). 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 227).

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 166). Macros may have a documentation string, but any `interactive` declaration is ignored since macros cannot be called interactively.

Macros often need to construct large list structures from a mixture of constants and nonconstant parts. To make this easier, use the '`' syntax (see Section 9.3 [Backquote], page 116). 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))
```

The body of a 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. The following specifications 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 264.

`(indent` *indent-spec*`)`
: Specify how to indent calls to this macro. See Section 13.6 [Indenting Macros], page 188, for more details.

`(doc-string` *number*`)`
: Specify which element of the macro is the documentation string, if any.

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 the secondary features listed above.

13.5 Common Problems Using Macros

Macro expansion can have counterintuitive consequences. This section describes some important consequences that can lead to trouble, and rules to follow to avoid trouble.

13.5.1 Wrong Time

The most common problem in writing macros is doing some of the real work prematurely—while expanding the macro, rather than in the expansion itself. For instance, one real package had this macro definition:

```
(defmacro my-set-buffer-multibyte (arg)
  (if (fboundp 'set-buffer-multibyte)
      (set-buffer-multibyte arg)))
```

With this erroneous macro definition, the program worked fine when interpreted but failed when compiled. This macro definition called `set-buffer-multibyte` during compilation, which was wrong, and then did nothing when the compiled package was run. The definition that the programmer really wanted was this:

```
(defmacro my-set-buffer-multibyte (arg)
  (if (fboundp 'set-buffer-multibyte)
      `(set-buffer-multibyte ,arg)))
```

This macro expands, if appropriate, into a call to `set-buffer-multibyte` that will be executed when the compiled program is actually run.

13.5.2 Evaluating Macro Arguments Repeatedly

When defining a macro you must pay attention to the number of times the arguments will be evaluated when the expansion is executed. The following macro (used to facilitate iteration) illustrates the problem. This macro allows us to write a "for" loop construct.

```
(defmacro for (var from init to final do &rest body)
  "Execute a simple \"for\" loop.
For example, (for i from 1 to 10 do (print i))."
  (list 'let (list (list var init))
        (cons 'while
              (cons (list '<= var final)
                    (append body (list (list 'inc var)))))))
⇒ for

(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.

13.5.3 Local Variables in Macro Expansions

The new definition of `for` has a new problem: it introduces a local variable named `max` which the user does not expect. This causes trouble in examples such as the following:

```
(let ((max 0))
  (for x from 0 to 10 do
    (let ((this (frob x)))
      (if (< max this)
          (setq max this)))))
```

The references to `max` inside the body of the `for`, which are supposed to refer to the user's binding of `max`, really access the binding made by `for`.

The way to correct this is to use an uninterned symbol instead of `max` (see Section 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.

13.5.4 Evaluating Macro Arguments in Expansion

Another problem can happen if the macro definition itself evaluates any of the macro argument expressions, such as by calling `eval` (see Section 9.4 [Eval], page 117). 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.

13.5.5 How Many Times is the Macro Expanded?

Occasionally problems result from the fact that a macro call is expanded each time it is evaluated in an interpreted function, but is expanded only once (during compilation) for a compiled function. If the macro definition has side effects, they will work differently depending on how many times the macro is expanded.

Therefore, you should avoid side effects in computation of the macro expansion, unless you really know what you are doing.

One special kind of side effect can't be avoided: constructing Lisp objects. Almost all macro expansions include constructed lists; that is the whole point of most macros. This is usually safe; there is just one case where you must be careful: when the object you construct is part of a quoted constant in the macro expansion.

If the macro is expanded just once, in compilation, then the object is constructed just once, during compilation. But in interpreted execution, the macro is expanded each time the macro call runs, and this means a new object is constructed each time.

In most clean Lisp code, this difference won't matter. It can matter only if you perform side-effects on the objects constructed by the macro definition. Thus, to avoid trouble,

avoid side effects on objects constructed by macro definitions. Here is an example of how such side effects can get you into trouble:

```
(defmacro empty-object ()
  (list 'quote (cons nil nil)))

(defun initialize (condition)
  (let ((object (empty-object)))
    (if condition
        (setcar object condition))
    object))
```

If `initialize` is interpreted, a new list `(nil)` is constructed each time `initialize` is called. Thus, no side effect survives between calls. If `initialize` is compiled, then the macro `empty-object` is expanded during compilation, producing a single "constant" `(nil)` that is reused and altered each time `initialize` is called.

One way to avoid pathological cases like this is to think of `empty-object` as a funny kind of constant, not as a memory allocation construct. You wouldn't use `setcar` on a constant such as `'(nil)`, so naturally you won't use it on `(empty-object)` either.

13.6 Indenting Macros

Within a macro definition, you can use the `declare` form (see Section 13.4 [Defining Macros], page 183) to specify how `TAB` should indent calls to the macro. An indentation specification is written 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 Customization Settings

This chapter describes how to declare customizable variables and customization groups for classifying them. We use the term *customization item* to include customizable variables, customization groups, as well as faces.

See Section 38.12.1 [Defining Faces], page 325, vol. 2, for the `defface` macro, which is used for declaring customizable faces.

14.1 Common Item Keywords

The customization declarations that we will describe in the next few sections (`defcustom`, `defgroup`, etc.) all accept keyword arguments for specifying various information. This section describes keywords that apply to all types of customization declarations.

All of these keywords, except `:tag`, can be used more than once in a given item. Each use of the keyword has an independent effect. The keyword `:tag` is an exception because any given item can only display one name.

`:tag` *`label`*

Use *label*, a string, instead of the item's name, to label the item in customization menus and buffers. **Don't use a tag which is substantially different from the item's real name; that would cause confusion.**

`:group` *`group`*

Put this customization item in group *group*. 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`’.

You can use this keyword more than once, to add multiple links.

`:load file`
: Load file *file* (a string) before displaying this customization item (see Chapter 15 [Loading], page 209). Loading is done with `load`, and only if the file is not already loaded.

`:require feature`
: Execute `(require 'feature)` when your saved customizations set the value of this item. *feature* should be a symbol.

The most common reason to use `:require` is when a variable enables a feature such as a minor mode, and just setting the variable won't have any effect unless the code which implements the mode is loaded.

`:version version`
: This keyword specifies that the item was first introduced in Emacs version *version*, or that its default value was changed in that version. The value *version* must be a string.

`:package-version '(package . version)`
: This keyword specifies that the item was first introduced in *package* version *version*, or that its meaning or default value was changed in that version. This keyword takes priority over `:version`.

package should be the official name of the package, as a symbol (e.g. `MH-E`). *version* should be a string. If the package *package* is released as part of Emacs, *package* and *version* should appear in the value of `customize-package-emacs-version-alist`.

Packages distributed as part of Emacs that use the `:package-version` keyword must also update the `customize-package-emacs-version-alist` variable.

`customize-package-emacs-version-alist` [Variable]

This alist provides a mapping for the versions of Emacs that are associated with versions of a package listed in the `:package-version` keyword. Its elements are:

`(`*`package`*` (`*`pversion`*` . `*`eversion`*`)...)`

For each *package*, which is a symbol, there are one or more elements that contain a package version *pversion* with an associated Emacs version *eversion*. These versions are strings. For example, the MH-E package updates this alist with the following:

```
(add-to-list 'customize-package-emacs-version-alist
             '(MH-E ("6.0" . "22.1") ("6.1" . "22.1") ("7.0" . "22.1")
                    ("7.1" . "22.1") ("7.2" . "22.1") ("7.3" . "22.1")
                    ("7.4" . "22.1") ("8.0" . "22.1")))
```

The value of *package* needs to be unique and it needs to match the *package* value appearing in the `:package-version` keyword. Since the user might see the value in an error message, a good choice is the official name of the package, such as MH-E or Gnus.

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 190), you can also use this keyword in `defgroup`:

`:prefix` *`prefix`*
: If the name of an item in the group starts with *prefix*, and the customizable variable `custom-unlispify-remove-prefixes` is non-`nil`, the item's tag will omit *prefix*. A group can have any number of prefixes.

`custom-unlispify-remove-prefixes` [User Option]
: If this variable is non-`nil`, the prefixes specified by a group's `:prefix` keyword are omitted from tag names, whenever the user customizes the group.

  The default value is `nil`, i.e. the prefix-discarding feature is disabled. This is because discarding prefixes often leads to confusing names for options and faces.

14.3 Defining Customization Variables

`defcustom` *option standard doc* [*keyword value*]. . . [Macro]
: This macro declares *option* as a user option (i.e. a customizable variable). You should not quote *option*.

  The argument *standard* is an expression that specifies the standard value for *option*. Evaluating the `defcustom` form evaluates *standard*, but does not necessarily 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.

  The argument *doc* specifies the documentation string for the variable.

  Every `defcustom` should specify `:group` at least once.

  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 141.

  If you put a `defcustom` in a pre-loaded Emacs Lisp file (see Section E.1 [Building Emacs], page 457, vol. 2), the standard value installed at dump time might be incorrect, e.g. because another variable that it depends on has not been assigned the right value yet. In that case, use `custom-reevaluate-setting`, described below, to re-evaluate the standard value after Emacs starts up.

`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 196, 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 when using the Customize interface. The function *setfunction* should take two arguments, a symbol (the option name) and the new value, and should do whatever is necessary to update the value properly for this option (which may not mean simply setting the option as a Lisp variable). The default for *setfunction* is `set-default`.

If you specify this keyword, the variable's documentation string should describe how to do the same job in hand-written Lisp code.

`: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 signal no error.

  These functions are meant for options defined in pre-loaded files, where the *standard* expression may signal an error because some required variable or function is not yet defined. The value normally gets updated in '`startup.el`', ignoring the value computed by `defcustom`. After startup, if one unsets the value and reevaluates the `defcustom`, the *standard* expression can be evaluated without error.

`:risky` *value*
: Set the variable's `risky-local-variable` property to *value* (see Section 11.11 [File Local Variables], page 156).

`:safe` *function*
: Set the variable's `safe-local-variable` property to *function* (see Section 11.11 [File Local Variables], page 156).

`:set-after` *variables*
: When setting variables according to saved customizations, make sure to set the variables *variables* before this one; i.e., delay setting this variable until after those others have been handled. Use `:set-after` if setting this variable won't work properly unless those other variables already have their intended values.

It is useful to specify the `:require` keyword for an option that "turns on" a certain feature. This causes Emacs to load the feature, if it is not already loaded, whenever the option is set. See Section 14.1 [Common Keywords], page 190. 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, `saved-value` to record the value saved by the user with the customization buffer, and `customized-value` to record the value set by the user with the customization buffer, but not saved. See Section 8.4 [Property Lists], page 106. These properties are lists, the car of which is an expression that evaluates to the value.

`custom-reevaluate-setting` *symbol* [Function]

> This function re-evaluates the standard value of *symbol*, which should be a user option declared via `defcustom`. If the variable was customized, this function re-evaluates the saved value instead. Then it sets the user option to that value (using the option's `:set` property if that is defined).
>
> This is useful for customizable options that are defined before their value could be computed correctly. For example, during startup Emacs calls this function for some user options that were defined in pre-loaded Emacs Lisp files, but whose initial values depend on information available only at run-time.

`custom-variable-p` *arg* [Function]

> This function returns non-`nil` if *arg* is a customizable variable. A customizable variable is either a variable that has a `standard-value` or `custom-autoload` property (usually meaning it was declared with `defcustom`), or an alias for another customizable variable.

`user-variable-p` *arg* [Function]

> This function is like `custom-variable-p`, except it also returns `t` if the first character of the variable's documentation string is the character '`*`'. That is an obsolete way of indicating a user option, so for most purposes you may consider `user-variable-p` as equivalent to `custom-variable-p`.

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 203).

Some type symbols do not use any arguments; those are called *simple types*. For a simple type, if you do not use any keyword-value pairs, you can omit the parentheses around the type symbol. For example just `string` as a customization type is equivalent to `(string)`.

All customization types are implemented as widgets; see Section "Introduction" in *The Emacs Widget Library*, for details.

14.4.1 Simple Types

This section describes all the simple customization types. For several of these customization types, the customization widget provides inline completion with *C-M-i* or *M-TAB*.

`sexp` The value may be any Lisp object that can be printed and read back. You can use `sexp` as a fall-back for any option, if you don't want to take the time to work out a more specific type to use.

`integer` The value must be an integer.

`number` The value must be a number (floating point or integer).

`float` The value must be a floating point number.

`string` The value must be a string. The customization buffer shows the string without delimiting '`"`' characters or '`\`' quotes.

`regexp` Like `string` except that the string must be a valid regular expression.

`character`
The value must be a character code. A character code is actually an integer, but this type shows the value by inserting the character in the buffer, rather than by showing the number.

`file` The value must be a file name. The widget provides completion.

`(file :must-match t)`
The value must be a file name for an existing file. The widget provides completion.

`directory`
The value must be a directory name. The widget provides completion.

`hook` The value must be a list of functions. This customization type is used for hook variables. You can use the `:options` keyword in a hook variable's `defcustom` to specify a list of functions recommended for use in the hook; See Section 14.3 [Variable Definitions], page 193.

`symbol` The value must be a symbol. It appears in the customization buffer as the symbol name. The widget provides completion.

`function` The value must be either a lambda expression or a function name. The widget provides completion for function names.

`variable` The value must be a variable name. The widget provides completion.

`face` The value must be a symbol which is a face name. The widget provides completion.

`boolean` The value is boolean—either `nil` or `t`. Note that by using `choice` and `const` together (see the next section), you can specify that the value must be `nil` or `t`, but also specify the text to describe each value in a way that fits the specific meaning of the alternative.

`coding-system`
: The value must be a coding-system name, and you can do completion with *`M-TAB`*.

`color` The value must be a valid color name. The widget provides completion for color names, as well as a sample and a button for selecting a color name from a list of color names shown in a '`*Colors*`' buffer.

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 CDR are displayed and edited separately, each according to their specified type.

`(list element-types...)`
: The value must be a list with exactly as many elements as the *element-types* given; and each element must fit the corresponding *element-type*.

  For example, `(list integer string function)` describes a list of three elements; the first element must be an integer, the second a string, and the third a function.

  In the customization buffer, each element is displayed and edited separately, according to the type specified for it.

`(group element-types...)`
: This works like `list` except for the formatting of text in the Custom buffer. `list` labels each element value with its tag; `group` does not.

`(vector element-types...)`
: Like `list` except that the value must be a vector instead of a list. The elements work the same as in `list`.

`(alist :key-type key-type :value-type value-type)`
: The value must be a list of cons-cells, the CAR of each cell representing a key of customization type *key-type*, and the CDR of the same cell representing a value

of customization type *value-type*. The user can add and delete key/value pairs, and edit both the key and the value of each pair.

If omitted, *key-type* and *value-type* default to `sexp`.

The user can add any key matching the specified key type, but you can give some keys a preferential treatment by specifying them with the `:options` (see Section 14.3 [Variable Definitions], page 193). The specified keys will always be shown in the customize buffer (together with a suitable value), with a checkbox to include or exclude or disable the key/value pair from the alist. The user will not be able to edit the keys specified by the `:options` keyword argument.

The argument to the `:options` keywords should be a list of specifications for reasonable keys in the alist. Ordinarily, they are simply atoms, which stand for themselves. For example:

```
:options '("foo" "bar" "baz")
```

specifies that there are three "known" keys, namely `"foo"`, `"bar"` and `"baz"`, which will always be shown first.

You may want to restrict the value type for specific keys, for example, the value associated with the `"bar"` key can only be an integer. You can specify this by using a list instead of an atom in the list. The first element will specify the key, like before, while the second element will specify the value type. For example:

```
:options '("foo" ("bar" integer) "baz")
```

Finally, you may want to change how the key is presented. By default, the key is simply shown as a `const`, since the user cannot change the special keys specified with the `:options` keyword. However, you may want to use a more specialized type for presenting the key, like `function-item` if you know it is a symbol with a function binding. This is done by using a customization type specification instead of a symbol for the key.

```
:options '("foo"
           ((function-item some-function) integer)
           "baz")
```

Many alists use lists with two elements, instead of cons cells. For example,

```
(defcustom list-alist
  '(("foo" 1) ("bar" 2) ("baz" 3))
  "Each element is a list of the form (KEY VALUE).")
```

instead of

```
(defcustom cons-alist
  '(("foo" . 1) ("bar" . 2) ("baz" . 3))
  "Each element is a cons-cell (KEY . VALUE).")
```

Because of the way lists are implemented on top of cons cells, you can treat `list-alist` in the example above as a cons cell alist, where the value type is a list with a single element containing the real value.

```
(defcustom list-alist '(("foo" 1) ("bar" 2) ("baz" 3))
  "Each element is a list of the form (KEY VALUE)."
  :type '(alist :value-type (group integer)))
```

The `group` widget is used here instead of `list` only because the formatting is better suited for the purpose.

Similarly, you can have alists with more values associated with each key, using variations of this trick:

```
(defcustom person-data '(("brian"  50 t)
                         ("dorith" 55 nil)
                         ("ken"    52 t))
  "Alist of basic info about people.
Each element has the form (NAME AGE MALE-FLAG)."
  :type '(alist :value-type (group integer boolean)))
```

`(plist :key-type` *`key-type`* `:value-type` *`value-type`*`)`

This customization type is similar to `alist` (see above), except that (i) the information is stored as a property list, (see Section 8.4 [Property Lists], page 106), and (ii) *key-type*, if omitted, defaults to `symbol` rather than `sexp`.

`(choice` *`alternative-types`*`...)`

The value must fit one of *alternative-types*. For example, `(choice integer string)` allows either an integer or a string.

In the customization buffer, the user selects an alternative using a menu, and can then edit the value in the usual way for that alternative.

Normally the strings in this menu are determined automatically from the choices; however, you can specify different strings for the menu by including the `:tag` keyword in the alternatives. For example, if an integer stands for a number of spaces, while a string is text to use verbatim, you might write the customization type this way,

```
(choice (integer :tag "Number of spaces")
        (string :tag "Literal text"))
```

so that the menu offers '`Number of spaces`' and '`Literal text`'.

In any alternative for which `nil` is not a valid value, other than a `const`, you should specify a valid default for that alternative using the `:value` keyword. See Section 14.4.4 [Type Keywords], page 203.

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:

`:tag tag`
: Use *tag* as the name of this alternative, for user communication purposes. This is useful for a type that appears inside of a `choice`.

`:match-alternatives criteria`
: Use *criteria* to match possible values. This is used only in `restricted-sexp`.

`:args argument-list`
: Use the elements of *argument-list* as the arguments of the type construct. For instance, `(const :args (foo))` is equivalent to `(const foo)`. You rarely need to write `:args` explicitly, because normally the arguments are recognized automatically as whatever follows the last keyword-value pair.

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` customization type. You use it by adding `:inline t` to a type specification which is contained in a `list` or `vector` specification.

Normally, each entry in a `list` or `vector` type specification describes a single element type. But when an entry contains `:inline t`, the value it matches is merged directly into the containing sequence. For example, if the entry matches a list with three elements, those become three elements of the overall sequence. This is analogous to '`,@`' in a backquote construct (see Section 9.3 [Backquote], page 116).

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`*
: Provide a default value.

  If `nil` is not a valid value for the alternative, then it is essential to specify a valid default with `:value`.

  If you use this for a type that appears as an alternative inside of `choice`; it specifies the default value to use, at first, if and when the user selects this alternative with the menu in the customization buffer.

  Of course, if the actual value of the option fits this alternative, it will appear showing the actual value, not *default*.

`:format` *`format-string`*
: This string will be inserted in the buffer to represent the value corresponding to the type. The following '`%`' escapes are available for use in *format-string*:

  '`%[`*`button`*`%]`'
  : Display the text *button* marked as a button. The `:action` attribute specifies what the button will do if the user invokes it; its value is a function which takes two arguments—the widget which the button appears in, and the event.

    There is no way to specify two different buttons with different actions.

  '`%{`*`sample`*`%}`'
  : Show *sample* in a special face specified by `:sample-face`.

  '`%v`'
  : Substitute the item's value. How the value is represented depends on the kind of item, and (for variables) on the customization type.

  '`%d`'
  : Substitute the item's documentation string.

  '`%h`'
  : Like '`%d`', but if the documentation string is more than one line, add 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.

`:validate` *`function`*
Specify a validation function for input. *function* takes a widget as an argument, and should return `nil` if the widget's current value is valid for the widget. Otherwise, it should return the widget containing the invalid data, and set that widget's `:error` property to a string explaining the error.

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.

14.5 Applying Customizations

The following functions are responsible for installing the user's customization settings for variables and faces, respectively. When the user invokes '`Save for future sessions`' in the Customize interface, that takes effect by writing a `custom-set-variables` and/or a `custom-set-faces` form into the custom file, to be evaluated the next time Emacs starts.

`custom-set-variables` **&rest** *args* [Function]

This function installs the variable customizations specified by *args*. Each argument in *args* should have the form

```
(var expression [now [request [comment]]])
```

var is a variable name (a symbol), and *expression* is an expression which evaluates to the desired customized value.

If the `defcustom` form for *var* has been evaluated prior to this `custom-set-variables` call, *expression* is immediately evaluated, and the variable's value is set to the result. Otherwise, *expression* is stored into the variable's `saved-value` property, to be evaluated when the relevant `defcustom` is called (usually when the library defining that variable is loaded into Emacs).

The *now*, *request*, and *comment* entries are for internal use only, and may be omitted. *now*, if non-`nil`, means to set the variable's value now, even if the variable's `defcustom` form has not been evaluated. *request* is a list of features to be loaded immediately (see Section 15.7 [Named Features], page 217). *comment* is a string describing the customization.

`custom-set-faces` **&rest** *args* [Function]

This function installs the face customizations specified by *args*. Each argument in *args* should have the form

```
(face spec [now [comment]])
```

face is a face name (a symbol), and *spec* is the customized face specification for that face (see Section 38.12.1 [Defining Faces], page 325, vol. 2).

The *now* and *comment* entries are for internal use only, and may be omitted. *now*, if non-`nil`, means to install the face specification now, even if the `defface` form has not been evaluated. *comment* is a string describing the customization.

14.6 Custom Themes

Custom themes are collections of settings that can be enabled or disabled as a unit. See Section "Custom Themes" in *The GNU Emacs Manual*. Each Custom theme is defined by an Emacs Lisp source file, which should follow the conventions described in this section. (Instead of writing a Custom theme by hand, you can also create one using a Customize-like interface; see Section "Creating Custom Themes" in *The GNU Emacs Manual*.)

A Custom theme file should be named '*foo*`-theme.el`', where *foo* is the theme name. The first Lisp form in the file should be a call to `deftheme`, and the last form should be a call to `provide-theme`.

`deftheme` *theme* **&optional** *doc* [Macro]

This macro declares *theme* (a symbol) as the name of a Custom theme. The optional argument *doc* should be a string describing the theme; this is the description shown when the user invokes the `describe-theme` command or types `?` in the '`*Custom Themes*`' buffer.

Two special theme names are disallowed (using them causes an error): `user` is a "dummy" theme that stores the user's direct customization settings, and `changed` is a "dummy" theme that stores changes made outside of the Customize system.

`provide-theme` *theme* [Macro]
This macro declares that the theme named *theme* has been fully specified.

In between `deftheme` and `provide-theme` are Lisp forms specifying the theme settings: usually a call to `custom-theme-set-variables` and/or a call to `custom-theme-set-faces`.

`custom-theme-set-variables` *theme* **&rest** *args* [Function]
This function specifies the Custom theme *theme*'s variable settings. *theme* should be a symbol. Each argument in *args* should be a list of the form

```
(var expression [now [request [comment]]])
```

where the list entries have the same meanings as in `custom-set-variables`. See Section 14.5 [Applying Customizations], page 206.

`custom-theme-set-faces` *theme* **&rest** *args* [Function]
This function specifies the Custom theme *theme*'s face settings. *theme* should be a symbol. Each argument in *args* should be a list of the form

```
(face spec [now [comment]])
```

where the list entries have the same meanings as in `custom-set-faces`. See Section 14.5 [Applying Customizations], page 206.

In theory, a theme file can also contain other Lisp forms, which would be evaluated when loading the theme, but that is "bad form". To protect against loading themes containing malicious code, Emacs displays the source file and asks for confirmation from the user before loading any non-built-in theme for the first time.

The following functions are useful for programmatically enabling and disabling themes:

`custom-theme-p` *theme* [Function]
This function return a non-`nil` value if *theme* (a symbol) is the name of a Custom theme (i.e. a Custom theme which has been loaded into Emacs, whether or not the theme is enabled). Otherwise, it returns `nil`.

`load-theme` *theme* **&optional** *no-confirm no-enable* [Command]
This function loads the Custom theme named *theme* from its source file, looking for the source file in the directories specified by the variable `custom-theme-load-path`. See Section "Custom Themes" in *The GNU Emacs Manual*. It also *enables* the theme (unless the optional argument *no-enable* is non-`nil`), causing its variable and face settings to take effect. It prompts the user for confirmation before loading the theme, unless the optional argument *no-confirm* is non-`nil`.

`enable-theme` *theme* [Command]
This function enables the Custom theme named *theme*. It signals an error if no such theme has been loaded.

`disable-theme` *theme* [Command]

This function disables the Custom theme named *theme*. The theme remains loaded, so that a subsequent call to `enable-theme` will re-enable it.

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. Such a file is also called a *Lisp library*.

The load functions evaluate all the expressions in a file just as the `eval-buffer` function evaluates all the expressions in a buffer. The difference is that the load functions read and evaluate the text in the file as found on disk, not the text in an Emacs buffer.

The loaded file must contain Lisp expressions, either as source code or as byte-compiled code. Each form in the file is called a *top-level form*. There is no special format for the forms in a loadable file; any form in a file may equally well be typed directly into a buffer and evaluated there. (Indeed, most code is tested this way.) Most often, the forms are function definitions and variable definitions.

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 213). `require` loads a file if it isn't already loaded (see Section 15.7 [Named Features], page 217). 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 file names like '`foo.el.el`' from being tried.

If the optional argument *must-suffix* is non-`nil`, then `load` insists that the file name used must end in either '`.el`' or '`.elc`' (possibly extended with a compression suffix), 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 211.

Whatever the name under which the file is eventually found, and the directory where Emacs found it, Emacs sets the value of the variable `load-file-name` to that file's name.

If you get a warning that '`foo.elc`' is older than '`foo.el`', it means you should consider recompiling '`foo.el`'. See Chapter 16 [Byte Compilation], page 223.

When loading a source file (not compiled), `load` performs character set translation just as Emacs would do when visiting the file. See Section 33.9 [Coding Systems], page 193, vol. 2.

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 for the way it reads its argument interactively. See Section "Lisp Libraries" in *The GNU Emacs Manual*.

`load-in-progress` [Variable]

This variable is non-`nil` if Emacs is in the process of loading a file, and it is `nil` otherwise.

`load-file-name` [Variable]

When Emacs is in the process of loading a file, this variable's value is the name of that file, as Emacs found it during the search described earlier in this section.

`load-read-function` [Variable]

> This variable specifies an alternate expression-reading function for `load` and `eval-region` to use instead of `read`. The function should accept one argument, just as `read` does.
>
> 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 118.

For information about how `load` is used in building Emacs, see Section E.1 [Building Emacs], page 457, vol. 2.

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` [Variable]

> The value of this variable is a list of directories to search when loading files with `load`. Each element is a string (which must be a directory name) or `nil` (which stands for the current working directory).

Each time Emacs starts up, it sets up the value of `load-path` in several steps. First, it initializes `load-path` to the directories specified by the environment variable `EMACSLOADPATH`, if that exists. The syntax of `EMACSLOADPATH` is the same as used for `PATH`; directory names are separated by '`:`' (or '`;`', on some operating systems), and '`.`' stands for the current default directory. Here is an example of how to set `EMACSLOADPATH` variable from `sh`:

```
export EMACSLOADPATH
EMACSLOADPATH=/home/foo/.emacs.d/lisp:/opt/emacs/lisp
```

Here is how to set it from `csh`:

```
setenv EMACSLOADPATH /home/foo/.emacs.d/lisp:/opt/emacs/lisp
```

If `EMACSLOADPATH` is not set (which is usually the case), Emacs initializes `load-path` with the following two directories:

```
"/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.

If you run Emacs from the directory where it was built—that is, an executable that has not been formally installed—Emacs puts two more directories in `load-path`. These are the `lisp` and `site-lisp` subdirectories of the main build directory. (Both are represented as absolute file names.)

Next, Emacs "expands" the initial list of directories in `load-path` by adding the subdirectories of those directories. Both immediate subdirectories and subdirectories multiple levels down are added. But it excludes subdirectories whose names do not start with a letter or digit, and subdirectories named '`RCS`' or '`CVS`', and subdirectories containing a file named '`.nosearch`'.

Next, Emacs adds any extra load directory that you specify using the '`-L`' command-line option (see Section "Action Arguments" in *The GNU Emacs Manual*). It also adds the directories where optional packages are installed, if any (see Section 40.1 [Packaging Basics], page 418, vol. 2).

It is common to add code to one's init file (see Section 39.1.2 [Init File], page 389, vol. 2) to add one or more directories to `load-path`. For example:

```
(push "~/.emacs.d/lisp" load-path)
```

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.

`locate-library` *library* **&optional** *nosuffix path interactive-call* [Command]

This command finds the precise file name for library *library*. It searches for the library in the same way `load` does, and the argument *nosuffix* has the same meaning as in `load`: don't add suffixes '`.elc`' or '`.el`' to the specified name *library*.

If the *path* is non-`nil`, that list of directories is used instead of `load-path`.

When `locate-library` is called from a program, it returns the file name as a string. When the user runs `locate-library` interactively, the argument *interactive-call* is `t`, and this tells `locate-library` to display the file name in the echo area.

`list-load-path-shadows` **&optional** *stringp* [Command]

This command shows a list of *shadowed* Emacs Lisp files. A shadowed file is one that will not normally be loaded, despite being in a directory on `load-path`, due to the existence of another similarly-named file in a directory earlier on `load-path`.

For instance, suppose `load-path` is set to

```
("/opt/emacs/site-lisp" "/usr/share/emacs/23.3/lisp")
```

and that both these directories contain a file named '`foo.el`'. Then `(require 'foo)` never loads the file in the second directory. Such a situation might indicate a problem in the way Emacs was installed.

When called from Lisp, this function prints a message listing the shadowed files, instead of displaying them in a buffer. If the optional argument `stringp` is non-`nil`, it instead returns the shadowed files as a string.

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 182, vol. 2). 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 33.9 [Coding Systems], page 193, vol. 2.

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 local variables section. 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 `?v`*`literal`*.

15.5 Autoload

The *autoload* facility allows you to register the existence of a function or macro, but put off loading the file that defines it. The first call to the function automatically reads the proper file, in order 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 451), `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`'. (The string that serves as the autoload cookie and the name of the file generated by `update-file-autoloads` can be changed from the above defaults, see below.) 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`'. The form following the magic comment is copied verbatim, *except* if it is one of the forms which the autoload facility handles specially (e.g. by conversion into an `autoload` call). The forms which are not copied verbatim are the following:

Definitions for function or function-like objects:
: `defun` and `defmacro`; also `defun*` and `defmacro*` (see Section "Argument Lists" in *CL Manual*), and `define-overloadable-function` (see the commentary in '`mode-local.el`').

Definitions for major or minor modes:
: `define-minor-mode`, `define-globalized-minor-mode`, `define-generic-mode`, `define-derived-mode`, `easy-mmode-define-minor-mode`, `easy-mmode-define-global-mode`, `define-compilation-mode`, and `define-global-minor-mode`.

Other definition types:
: `defcustom`, `defgroup`, `defclass` (see Section "Top" in *EIEIO*), and `define-skeleton` (see the commentary in '`skeleton.el`').

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 457, vol. 2. 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 457) display it.

If you write a function definition with an unusual macro that is not one of the known and recognized function definition methods, use of an ordinary magic autoload comment would copy the whole definition into loaddefs.el. That is not desirable. You can put the desired autoload call into loaddefs.el instead by writing this:

```
;;;###autoload (autoload 'foo "myfile")
(mydefunmacro foo
  ...)
```

You can use a non-default string as the autoload cookie and have the corresponding autoload calls written into a file whose name is different from the default 'loaddefs.el'. Emacs provides two variables to control this:

generate-autoload-cookie [Variable]

The value of this variable should be a string whose syntax is a Lisp comment. *M-x update-file-autoloads* copies the Lisp form that follows the cookie into the autoload file it generates. The default value of this variable is ";;;###autoload".

generated-autoload-file [Variable]

The value of this variable names an Emacs Lisp file where the autoload calls should go. The default value is 'loaddefs.el', but you can override that, e.g., in the "Local Variables" section of a '.el' file (see Section 11.11 [File Local Variables], page 156). The autoload file is assumed to contain a trailer starting with a formfeed character.

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 141.)

The simplest way to add an element to an alist is like this:

```
(push '(leif-mode " Leif") minor-mode-alist)
```

But this would add multiple elements if the library is reloaded. To avoid the problem, use `add-to-list` (see Section 5.5 [List Variables], page 71):

```
(add-to-list 'minor-mode-alist '(leif-mode " Leif"))
```

Occasionally you will want to test explicitly whether a library has already been loaded. If the library uses `provide` to provide a named feature, you can use `featurep` earlier in the file to test whether the `provide` call has been executed before (see Section 15.7 [Named Features], page 217). Alternatively, you could use something like this:

```
(defvar foo-was-loaded nil)

(unless foo-was-loaded
  execute-first-time-only
  (setq foo-was-loaded t))
```

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 '`idlwave.el`', the definition for `idlwave-complete-filename` includes the following code:

```
(defun idlwave-complete-filename ()
  "Use the comint stuff to complete a file name."
   (require 'comint)
   (let* ((comint-file-name-chars "~/A-Za-z0-9+_.$#%={}\\-")
          (comint-completion-addsuffix nil)
          ...)
       (comint-dynamic-complete-filename)))
```

The expression `(require 'comint)` loads the file '`comint.el`' if it has not yet been loaded, ensuring that `comint-dynamic-complete-filename` is defined. Features are normally named after the files that provide them, so that `require` need not be given the file name.

(Note that it is important that the `require` statement be outside the body of the `let`. Loading a library while its variables are let-bound can have unintended consequences, namely the variables becoming unbound after the let exits.)

The '`comint.el`' file contains the following top-level expression:

```
(provide 'comint)
```

This adds `comint` to the global `features` list, so that `(require 'comint)` will henceforth know that nothing needs to be done.

When `require` is used at top level in a file, it takes effect when you byte-compile that file (see Chapter 16 [Byte Compilation], page 223) 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 if not already in *features* then to add *feature* to the front of that list and call any `eval-after-load` code waiting for it (see Section 15.10 [Hooks for Loading], page 221). 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 37.17.3 [Network Feature Testing], page 288, vol. 2, 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 213.

`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`, `defvar`, or `defface`, that specifies function definition, variable definition, or face definition only.

The value is normally an absolute file name. It can also be `nil`, if the definition is not associated with any file. If *symbol* specifies an autoloaded function, the value can be a relative file name without extension.

The basis for `symbol-file` is the data in the variable `load-history`.

`load-history` [Variable]

The value of this variable is an alist that associates the names of loaded library files with the names of the functions and variables they defined, as well as the features they provided or required.

Each element in this alist describes one loaded library (including libraries that are preloaded at startup). It is a list whose CAR is the absolute file name of the library (a string). The rest of the list elements have these forms:

`var` The symbol *var* was defined as a variable.

`(defun . fun)`
The function *fun* was defined.

`(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.

`(defface . face)`
The face *face* was defined.

`(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 9.4 [Eval], page 117.

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`, as well as `auto-mode-alist`. This is to prevent Emacs from ceasing to function because important hooks refer to functions that are no longer defined.

Standard unloading activities also undoes ELP profiling of functions in that library, unprovides any features provided by the library, and cancels timers held in variables defined by the library.

If these measures are not sufficient to prevent malfunction, a library can define an explicit unloader named `feature-unload-function`. If that symbol is defined as a function, `unload-feature` calls it with no arguments before doing anything else.

It can do whatever is appropriate to unload the library. If it returns `nil`, `unload-feature` proceeds to take the normal unload actions. Otherwise it considers the job to be done.

Ordinarily, `unload-feature` refuses to unload a library on which other loaded libraries depend. (A library *a* depends on library *b* if *a* contains a `require` for *b*.) If the optional argument *force* is non-`nil`, dependencies are ignored and you can unload any library.

The `unload-feature` function is written in Lisp; its actions are based on the variable `load-history`.

`unload-feature-special-hooks` [Variable]

This variable holds a list of hooks to be scanned before unloading a library, to remove functions defined in the library.

15.10 Hooks for Loading

You can ask for code to be executed each time Emacs loads a library, by using the variable `after-load-functions`:

`after-load-functions` [Variable]

This abnormal hook is run after loading a file. Each function in the hook is called with a single argument, the absolute filename of the file that was just loaded.

If you want code to be executed when a *particular* library is loaded, use the function `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-edebug-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 at the end of any file where `(provide library)` is called.

An error in *form* does not undo the load, but does prevent execution of the rest of *form*.

Normally, well-designed Lisp programs should not use `eval-after-load`. If you need to examine and set the variables defined in another library (those meant for outside use), you can do it immediately—there is no need to wait until the library is loaded. If you need to call functions defined by that library, you should load the library, preferably with `require` (see Section 15.7 [Named Features], page 217).

`after-load-alist` [Variable]

This variable stores an alist built by `eval-after-load`, containing the expressions to evaluate when certain 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 or feature name of the library being loaded.

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.

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; -*-
```

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 the time, in seconds, to run N iterations of a loop."
  (let ((t1 (float-time)))
    (while (> (setq n (1- n)) 0))
    (- (float-time) t1)))
⇒ silly-loop

(silly-loop 50000000)
⇒ 10.235304117202759

(byte-compile 'silly-loop)
⇒ [Compiled code not shown]

(silly-loop 50000000)
⇒ 3.705854892730713
```

In this example, the interpreted code required 10 seconds to run, whereas the byte-compiled code required less than 4 seconds. These results are representative, but actual results may vary.

16.2 Byte-Compilation Functions

You can byte-compile an individual function or macro definition with the `byte-compile` function. You can compile a whole file with `byte-compile-file`, or several files with `byte-recompile-directory` or `batch-byte-compile`.

Sometimes, the byte compiler produces warning and/or error messages (see Section 16.6 [Compiler Errors], page 228, for details). These messages are recorded in a buffer called

'`*Compile-Log*`', which uses Compilation mode. See Section "Compilation Mode" in *The GNU Emacs Manual*.

Be careful when writing macro calls in files that you intend to byte-compile. Since macro calls are expanded when they are compiled, the macros need to be loaded into Emacs or the byte compiler will not do the right thing. The usual way to handle this is with `require` forms which specify the files containing the needed macro definitions (see Section 15.7 [Named Features], page 217). Normally, the byte compiler does not evaluate the code that it is compiling, but it handles `require` forms specially, by loading the specified libraries. To avoid loading the macro definition files when someone *runs* the compiled program, write `eval-when-compile` around the `require` calls (see Section 16.5 [Eval During Compile], page 227). For more details, See Section 13.3 [Compiling Macros], page 182.

Inline (`defsubst`) functions are less troublesome; if you compile a call to such a function before its definition is known, the call will still work right, it will just run slower.

`byte-compile` *symbol* [Function]

This function byte-compiles the function definition of *symbol*, replacing the previous definition with the compiled one. The function definition of *symbol* must be the actual code for the function; `byte-compile` does not handle function indirection. The return value is the byte-code function object which is the compiled definition of *symbol* (see Section 16.7 [Byte-Code Objects], page 229).

```
(defun factorial (integer)
  "Compute factorial of INTEGER."
  (if (= 1 integer) 1
    (* integer (factorial (1- integer)))))
⇒ factorial

(byte-compile 'factorial)
⇒
#[(integer)
  "^H\301U\203^H^@\301\207\302^H\303^HS!\"\207"
  [integer 1 * factorial]
  4 "Compute factorial of INTEGER."]
```

If *symbol*'s definition is a byte-code function object, `byte-compile` does nothing and returns `nil`. It does not "compile the symbol's definition again", since the original (non-compiled) code has already been replaced in the symbol's function cell by the byte-compiled code.

The argument to `byte-compile` can also be a `lambda` expression. In that case, the function returns the corresponding compiled code but does not store it anywhere.

`compile-defun` **&optional** *arg* [Command]

This command reads the defun containing point, compiles it, and evaluates the result. If you use this on a defun that is actually a function definition, the effect is to install a compiled version of that function.

`compile-defun` normally displays the result of evaluation in the echo area, but if *arg* is non-`nil`, it inserts the result in the current buffer after the form it 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
```

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.

These problems normally occur only if you build Emacs yourself and use it from the directory where you built it, and you happen to edit and/or recompile the Lisp source files. They can be easily cured by reloading each file after recompiling it.

The dynamic documentation string feature writes compiled files that use a special Lisp reader construct, '`#@`*count*'. This construct skips the next *count* characters. It also uses the '`#$`' construct, which stands for "the name of this file, as a string". 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.

You can disable the dynamic documentation string 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` [User Option]

If this is non-`nil`, the byte compiler generates compiled files that are set up for dynamic loading of documentation strings.

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))
```

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.

When an error is due to invalid syntax in the program, the byte compiler might get confused about the errors' exact location. One way to investigate is to switch to the buffer '` *Compiler Input*`'. (This buffer name starts with a space, so it does not show up in *`M-x list-buffers`*.) This buffer contains the program being compiled, and point shows how far the byte compiler was able to read; the cause of the error might be nearby. See Section 18.3 [Syntax Errors], page 271, for some tips for locating syntax errors.

When the byte compiler warns about functions that were used but not defined, it always reports the line number for the end of the file, not the locations where the missing functions were called. To find the latter, 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.)

You can tell the compiler that a function is defined using `declare-function` (see Section 12.12 [Declaring Functions], page 178). Likewise, you can tell the compiler that a variable is defined using `defvar` with no initial value.

You can suppress the compiler warning for a specific use of an undefined 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 and all compiler warnings within a certain expression 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, to avoid missing possible warnings other than one you intend to suppress.

More precise control of warnings is possible by setting the variable `byte-compile-warnings`.

16.7 Byte-Code Function Objects

Byte-compiled functions have a special data type: they are *byte-code function objects*. Whenever such an object appears as a function to be called, Emacs uses the byte-code interpreter to execute the byte-code.

Internally, a byte-code function object is much like a vector; its elements can be accessed using `aref`. Its printed representation is like that for a vector, with an additional '`#`' before the opening '`[`'. It must have at least four elements; there is no maximum number, but only the first six elements have any normal use. They are:

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 452).
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^@\301^P\302^H[!\207"
  [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).

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 human-readable form.

The byte-code interpreter is implemented as a simple stack machine. It pushes values onto a stack of its own, then pops them off to use them in calculations whose results are themselves pushed back on the stack. When a byte-code function returns, it pops a value off the stack and returns it as the value of the function.

In addition to the stack, byte-code functions can use, bind, and set ordinary Lisp variables, by transferring values between variables and the stack.

`disassemble` *object* **&optional** *buffer-or-name* [Command]

This command displays the disassembled code for *object*. In interactive use, or if *buffer-or-name* is `nil` or omitted, the output goes in a buffer named '`*Disassemble*`'. If *buffer-or-name* is non-`nil`, it must be a buffer or the name of an existing buffer. Then the output goes there, at point, and point is left before the output.

The argument *object* can be a function name, a lambda expression 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`.

```
(defun factorial (integer)
  "Compute factorial of an integer."
  (if (= 1 integer) 1
    (* integer (factorial (1- integer)))))
     ⇒ factorial

(factorial 4)
     ⇒ 24
```

```
(disassemble 'factorial)
     ⊣ byte-code for factorial:
 doc: Compute factorial of an integer.
 args: (integer)

0   varref   integer        ; Get the value of integer and
                            ;   push it onto the stack.
1   constant 1              ; Push 1 onto stack.
2   eqlsign                 ; Pop top two values off stack, compare
                            ;   them, and push result onto stack.
3   goto-if-nil 1           ; Pop and test top of stack;
                            ;   if nil, go to 1, else continue.
6   constant 1              ; Push 1 onto top of stack.
7   return                  ; Return the top element of the stack.
8:1 varref   integer        ; Push value of integer onto stack.
9   constant factorial      ; Push factorial onto stack.
10  varref   integer        ; Push value of integer onto stack.
11  sub1                    ; Pop integer, decrement value,
                            ;   push new value onto stack.
12  call     1              ; Call function factorial using first
                            ;   (i.e. top) stack element as argument;
                            ;   push returned value onto stack.
13 mult                     ; Pop top two values off stack, multiply
                            ;   them, and push result onto stack.
14 return                   ; Return the top element of the stack.
```

The silly-loop function is somewhat more complex:

```
(defun silly-loop (n)
  "Return time before and after N iterations of a loop."
  (let ((t1 (current-time-string)))
    (while (> (setq n (1- n))
              0))
    (list t1 (current-time-string))))
     ⇒ silly-loop

(disassemble 'silly-loop)
     ⊣ byte-code for silly-loop:
 doc: Return time before and after N iterations of a loop.
 args: (n)

0   constant current-time-string  ; Push current-time-string
                                  ;   onto top of stack.
1   call     0              ; Call current-time-string with no
                            ;   argument, push result onto stack.
2   varbind  t1             ; Pop stack and bind t1 to popped value.
```

```
3:1 varref   n               ; Get value of n from the environment
                             ;    and push the value on the stack.
4   sub1                     ; Subtract 1 from top of stack.
5   dup                      ; Duplicate top of stack; i.e. copy the top
                             ;    of the stack and push copy onto 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-not-nil 1        ; Goto 1 if n > 0
                             ;    (this continues the while loop)
                             ;    else continue.
12  varref   t1              ; Push value of t1 onto stack.
13  constant current-time-string  ; Push current-time-string
                                  ;    onto the top of the stack.
14  call     0               ; Call current-time-string again.
15  unbind   1               ; Unbind t1 in local environment.
16  list2                    ; Pop top two elements off stack, create a
                             ;    list of them, and push it onto stack.
17  return                   ; Return value of the top of stack.
```

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*, each of which can be separately defined and then *enabled* or *disabled*. 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.

Advice is useful for altering the behavior of existing calls to an existing function. If you want the new behavior for new function calls or new key bindings, you should define a new function or command, and have it use the existing function as a subroutine.

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 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, Emacs's own source files should not put advice on functions in Emacs. There are currently a few exceptions to this convention, but we aim to correct them.

Unless you know what you are doing, do *not* advise a primitive (see Section 12.1 [What Is a Function], page 163). Some primitives are used by the advice mechanism; advising them could cause an infinite recursion. Also, many primitives 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.

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 17.9 [Combined Definition], page 242). 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 240, 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 135.

`compile`
: Compile the combined definition that is used to run the advice. This flag is ignored unless `activate` is also specified. See Section 17.9 [Combined Definition], page 242.

`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 239.

`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 240.

  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 all pieces of 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; if *protected* is non-`nil`, the advice is protected against non-local exits (see Section 17.2 [Defining Advice], page 234), and if *enabled* is `nil` the advice is initially disabled (see Section 17.6 [Enabling Advice], page 239). *definition* should have the form

```
(advice . lambda)
```

where *lambda* is a lambda expression; this lambda expression is called in order to perform the advice. See Section 12.2 [Lambda Expressions], page 165.

If the *function* argument to `ad-add-advice` 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. 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`; or you can activate the advice separately 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 239.) 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 239) 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 240), 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, preactivation 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 preactivation.

Here are some symptoms that can indicate that a preactivation 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 preactivated advice works properly even if the function itself is not defined until later; however, the function needs to be defined when you *compile* the preactivated advice.

There is no elegant way to find out why preactivated 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 preactivated advice was used, while other values give some information about why they were considered inappropriate.

Warning: There is one known case that can make preactivation 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 their (zero-based) 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 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 several ways to find and investigate problems in an Emacs Lisp program.

- If a problem occurs when you run the program, you can use the built-in Emacs Lisp debugger to suspend the Lisp evaluator, and examine and/or alter its internal state.
- You can use Edebug, a source-level debugger for Emacs Lisp.
- If a syntactic problem is preventing Lisp from even reading the program, you can locate it using Lisp editing commands.
- You can look at the error and warning messages produced by the byte compiler when it compiles the program. See Section 16.6 [Compiler Errors], page 228.
- You can use the Testcover package to perform coverage testing on the program.
- You can use the ERT package to write regression tests for the program. See Section "Top" in *ERT: Emacs Lisp Regression Testing*.

Other useful tools for debugging input and output problems are the dribble file (see Section 39.12 [Terminal Input], page 409, vol. 2) and the `open-termscript` function (see Section 39.13 [Terminal Output], page 411, vol. 2).

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 21.13 [Recursive Editing], page 355.

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 signal Lisp errors when invoked inappropriately, and during ordinary editing it would be very inconvenient to enter the debugger each time this happens. So if you want errors to enter the debugger, set the variable `debug-on-error` to non-`nil`. (The command `toggle-debug-on-error` provides an easy way to do this.)

`debug-on-error` [User Option]

> This variable determines whether the debugger is called when an error is signaled and not handled. If `debug-on-error` is `t`, all kinds of errors call the debugger, except those listed in `debug-ignored-errors` (see below). If it is `nil`, none call the debugger.
>
> The value can also be a list of error conditions (see Section 10.5.3.1 [Signaling Errors], page 128). Then the debugger is called only for error conditions in this list (except those also listed in `debug-ignored-errors`). For example, if you set `debug-on-error` to the list `(void-variable)`, the debugger is only called for errors about a variable that has no value.

Note that `eval-expression-debug-on-error` overrides this variable in some cases; see below.

When this variable is non-`nil`, Emacs does not create an error handler around process filter functions and sentinels. Therefore, errors in these functions also invoke the debugger. See Chapter 37 [Processes], page 257, vol. 2.

`debug-ignored-errors` [User Option]

This variable specifies errors which should not enter the debugger, regardless of the value of `debug-on-error`. Its value is a list of error condition symbols and/or regular expressions. If the error has any of those condition symbols, or if the error message matches any of the regular expressions, then that error does not enter the debugger.

The normal value of this variable 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 may try changing this list to debug the error. The easiest way is usually to set `debug-ignored-errors` to `nil`.

`eval-expression-debug-on-error` [User Option]

If this variable has a non-`nil` value (the default), running the command `eval-expression` causes `debug-on-error` to be temporarily bound to to `t`. See Section "Evaluating Emacs-Lisp Expressions" in *The GNU Emacs Manual*.

If `eval-expression-debug-on-error` is `nil`, then the value of `debug-on-error` is not changed during `eval-expression`.

`debug-on-signal` [Variable]

Normally, errors caught by `condition-case` never invoke the debugger. The `condition-case` gets a chance to handle the error before the debugger gets a chance.

If you change `debug-on-signal` to a non-`nil` value, the debugger gets the first chance at every error, regardless of the presence of `condition-case`. (To invoke the debugger, the error must still fulfill the criteria specified by `debug-on-error` and `debug-ignored-errors`.)

Warning: Setting this variable to non-`nil` may have annoying effects. Various parts of Emacs catch errors in the normal course of affairs, and you may not even realize that errors happen there. If you need to debug code wrapped in `condition-case`, consider using `condition-case-unless-debug` (see Section 10.5.3.3 [Handling Errors], page 130).

`debug-on-event` [User Option]

If you set `debug-on-event` to a special event (see Section 21.9 [Special Events], page 350), Emacs will try to enter the debugger as soon as it receives this event, bypassing `special-event-map`. At present, the only supported values correspond to the signals `SIGUSR1` and `SIGUSR2` (this is the default). This can be helpful when `inhibit-quit` is set and Emacs is not otherwise responding.

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.

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*. See Section 21.11 [Quitting], page 351.

Ordinary quitting gives no information about why the program was looping. To get more information, you can set the variable `debug-on-quit` to non-`nil`. Once you have the debugger running in the middle of the infinite loop, you can proceed from the debugger using the stepping commands. If you step through the entire loop, you may get enough information to solve the problem.

Quitting with *C-g* is not considered an error, and `debug-on-error` has no effect on the handling of *C-g*. Likewise, `debug-on-quit` has no effect on errors.

`debug-on-quit` [User Option]

> This variable determines whether the debugger is called when `quit` is signaled and not handled. If `debug-on-quit` is non-`nil`, then the debugger is called whenever you quit (that is, type *C-g*). If `debug-on-quit` is `nil` (the default), then the debugger is not called when you quit.

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 120).

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 21.13 [Recursive Editing], page 355) 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.

When the debugger has been entered, the `debug-on-error` variable is temporarily set according to `eval-expression-debug-on-error`. If the latter variable is non-`nil`, `debug-on-error` will temporarily be set to `t`. This means that any further errors that occur while doing a debugging session will (by default) trigger another backtrace. If this is not want you want, you can either set `eval-expression-debug-on-error` to `nil`, or set `debug-on-error` to `nil` in `debugger-mode-hook`.

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 with the mouse 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. This resumes execution of the program as if the debugger had never been entered (aside from any side-effects that you caused by changing variable values or data structures while inside the debugger).

`d`
: Continue execution, but enter the debugger the next time any Lisp function is called. This allows you to step through the subexpressions of an expression, seeing what values the subexpressions compute, and what else they do.

: The stack frame made for the function call which enters the debugger in this way will be flagged automatically so that the debugger will be called again when the frame is exited. You can use the `u` command to cancel this flag.

`b`
: Flag the current frame so that the debugger will be entered when the frame is exited. Frames flagged in this way are marked with stars in the backtrace buffer.

u Don't enter the debugger when the current frame is exited. This cancels a *b* command on that frame. The visible effect is to remove the star from the line in the backtrace buffer.

j Flag the current frame like *b*. Then continue execution like *c*, but temporarily disable break-on-entry for all functions that are set up to do so by `debug-on-entry`.

e Read a Lisp expression in the minibuffer, evaluate it, 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* [Command]

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 *debugger-args* is that `debug` displays the rest of its arguments at the top of the '`*Backtrace*`' buffer, so that the user can see them. Except as described below, this is the *only* way these arguments are used.

However, certain values for first argument to `debug` have a special significance. (Normally, these values are used only by the internals of Emacs, and not by programmers calling `debug`.) Here is a table of these special values:

`lambda`
: A first argument of `lambda` means `debug` was called because of entry to a function when `debug-on-next-call` was non-`nil`. The debugger displays '`Debugger entered--entering a function:`' as a line of text at the top of the buffer.

`debug`
: `debug` as first argument means `debug` was called because of entry to a function that was set to debug on entry. The debugger displays the string '`Debugger entered--entering a function:`', just as in the `lambda` case. It also marks the stack frame for that function so that it will invoke the debugger when exited.

`t`
: When the first argument is `t`, this indicates a call to `debug` due to evaluation of a function call form when `debug-on-next-call` is non-`nil`. The debugger displays '`Debugger entered--beginning evaluation of function call form:`' as the top line in the buffer.

`exit`
: When the first argument is `exit`, it indicates the exit of a stack frame previously marked to invoke the debugger on exit. The second argument given to `debug` in this case is the value being returned from the frame. The debugger displays '`Debugger entered--returning value:`' in the top line of the buffer, followed by the value being returned.

`error`
: When the first argument is `error`, the debugger indicates that it is being entered because an error or `quit` was signaled and not handled, by displaying '`Debugger entered--Lisp error:`' followed by the error signaled and any arguments to `signal`. For example,

```
(let ((debug-on-error t))
  (/ 1 0))

------ Buffer: *Backtrace* ------
Debugger entered--Lisp error: (arith-error)
  /(1 0)
...
------ Buffer: *Backtrace* ------
```

If an error was signaled, presumably the variable `debug-on-error` is non-`nil`. If `quit` was signaled, then presumably the variable `debug-on-quit` is non-`nil`.

`nil`
: Use `nil` as the first of the *debugger-args* when you want to enter the debugger explicitly. The rest of the *debugger-args* are printed on the top line of the buffer. You can use this feature to display messages—for example, to remind yourself of the conditions under which `debug` is called.

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 248).

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 information on function calls and returns.
- Stop when an error occurs.
- Display a backtrace, omitting Edebug's own frames.
- Specify argument evaluation for macros and defining forms.
- Obtain rudimentary coverage testing and frequency counts.

The first three sections below should tell you enough about Edebug to start using it.

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 253, 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 253.

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 it is not already instrumented. This is possible only if Edebug knows where to find the source for that function; for this reason, after loading Edebug, `eval-region` records the position of every definition it evaluates, even if not instrumenting it. See also the *i* command (see Section 18.2.4 [Jumping], page 255), which steps into the call after instrumenting the function.

Edebug knows how to instrument all the standard special forms, `interactive` forms with an expression argument, anonymous lambda expressions, and other defining forms. However, Edebug cannot determine on its own what a user-defined macro will do with the arguments of a macro call, so you must provide that information using Edebug specifications; for details, see Section 18.2.15 [Edebug and Macros], page 264.

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 259, 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.

S	Stop: don't execute any more of the program, but wait for more Edebug commands (`edebug-stop`).
`SPC`	Step: stop at the next stop point encountered (`edebug-step-mode`).
n	Next: stop at the next stop point encountered after an expression (`edebug-next-mode`). Also see `edebug-forward-sexp` in Section 18.2.4 [Jumping], page 255.
t	Trace: pause (normally one second) at each Edebug stop point (`edebug-trace-mode`).
T	Rapid trace: update the display at each stop point, but don't actually pause (`edebug-Trace-fast-mode`).
g	Go: run until the next breakpoint (`edebug-go-mode`). See Section 18.2.6.1 [Breakpoints], page 256.
c	Continue: pause one second at each breakpoint, and then continue (`edebug-continue-mode`).
C	Rapid continue: move point to each breakpoint, but don't pause (`edebug-Continue-fast-mode`).
G	Go non-stop: ignore breakpoints (`edebug-Go-nonstop-mode`). You can still stop the program by typing *S*, or any editing command.

In general, the execution modes earlier in the above list run the program more slowly or stop sooner than the modes later in the list.

While executing or tracing, you can interrupt the execution by typing any Edebug command. Edebug stops the program at the next stop point and then executes the command you typed. For example, typing *t* during execution switches to trace mode at the next stop point. You can use *S* to stop execution without doing anything else.

If your function happens to read input, a character you type intending to interrupt execution may be read by the function instead. You can avoid such unintended results by paying attention to when your program wants input.

Keyboard macros containing the commands in this section do not completely work: exiting from Edebug, to resume the program, loses track of the keyboard macro. This is not easy to fix. Also, defining or executing a keyboard macro outside of Edebug does not affect commands inside Edebug. This is usually an advantage. See also the `edebug-continue-kbd-macro` option in Section 18.2.16 [Edebug Options], page 269.

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 269). 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 or continue 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 256, for the details on breakpoints.

These commands may fail to work as expected in case of nonlocal exit, as that can bypass the temporary breakpoint where you expected the program to stop.

h	Proceed to the stop point near where point is (`edebug-goto-here`).
f	Run the program for one expression (`edebug-forward-sexp`).
o	Run the program until the end of the containing sexp (`edebug-step-out`).
i	Step into the function or macro called by the form after point (`edebug-step-in`).

The *h* command proceeds to the stop point at or after the current location of point, using a temporary breakpoint.

The *f* command runs the program forward over one expression. More precisely, it sets a temporary breakpoint at the position that `forward-sexp` would reach, then executes in go mode so that the program will stop at breakpoints.

With a prefix argument *n*, the temporary breakpoint is placed *n* sexps beyond point. If the containing list ends before *n* more elements, then the place to stop is after the containing expression.

You must check that the position `forward-sexp` finds is a place that the program will really get to. In `cond`, for example, this may not be true.

For flexibility, the *f* command does `forward-sexp` starting at point, rather than at the stop point. If you want to execute one expression *from the current stop point*, first type *w* (`edebug-where`) to move point there, and then type *f*.

The *o* command continues “out of” an expression. It places a temporary breakpoint at the end of the sexp containing point. If the containing sexp is a function definition itself, *o* continues until just before the last sexp in the definition. If that is where you are now, it returns from the function and then stops. In other words, this command does not exit the currently executing function unless you are positioned after the last sexp.

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 (`edebug-top-level-nonstop`).

r Redisplay the most recently known expression result in the echo area (`edebug-previous-result`).

d Display a backtrace, excluding Edebug's own functions for clarity (`edebug-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-]*. 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 252. 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 269.

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 255).

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 263, 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 (`eval-expression`).

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`': `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* (`edebug-update-eval-list`) rebuilds the evaluation list, scanning the buffer and using the first expression of each group. (The idea is that the second expression of the group is the value previously computed and displayed.)

Each entry to Edebug redisplays the evaluation list by inserting each expression in the buffer, followed by its current value. It also inserts comment lines so that each expression becomes its own group. Thus, if you type *C-c C-u* again without changing the buffer text, the evaluation list is effectively unchanged.

If an error occurs during an evaluation from the evaluation list, the error message is displayed in a string as if it were the result. Therefore, expressions using variables that are not currently valid do not interrupt your debugging.

Here is an example of what the evaluation list window looks like after several expressions have been added to it:

```
(current-buffer)
#<buffer *scratch*>
;---------------------------------------------------------------
(selected-window)
#<window 16 on *scratch*>
;---------------------------------------------------------------
(point)
196
;---------------------------------------------------------------
bad-var
"Symbol’s value as variable is void: bad-var"
;---------------------------------------------------------------
(recursion-depth)
0
;---------------------------------------------------------------
this-command
eval-last-sexp
;---------------------------------------------------------------
```

To delete a group, move point into it and type *C-c C-d*, or simply delete the text for the group and update the evaluation list with *C-c C-u*. To add a new expression to the evaluation list, insert the expression at a suitable place, insert a new comment line, then type *C-c C-u*. You need not insert dashes in the comment line—its contents don’t matter.

After selecting ‘`*edebug*`’, you can return to the source code buffer with *C-c C-w*. The ‘`*edebug*`’ buffer is killed when you continue execution, and recreated next time it is needed.

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 the values of the variables `edebug-print-length` and `edebug-print-level` (so long as they have non-`nil` values). See Section 19.6 [Output Variables], page 282.

`edebug-print-length` [User Option]

If non-`nil`, Edebug binds `print-length` to this value while printing results. The default value is `50`.

`edebug-print-level` [User Option]
: If non-`nil`, Edebug binds `print-level` to this value while printing results. The default value is `50`.

You can also print circular structures and structures that share elements more informatively by binding `print-circle` to a non-`nil` value.

Here is an example of code that creates a circular structure:

```
(setq a '(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 253).

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 executions of an instrumented function, even if the execution mode is Go-nonstop, and regardless of whether coverage testing is enabled.

Use *C-x X* `=` (`edebug-display-freq-count`) to display both the coverage information and the frequency counts for a definition. Just `=` (`edebug-temp-display-freq-count`) displays the same information temporarily, only until you type another key.

`edebug-display-freq-count` [Command]

> This command displays the frequency count data for each line of the current definition.
>
> It inserts frequency counts as comment lines after each line of code. You can undo all insertions with one `undo` command. The counts appear under the '`(`' before an expression or the '`)`' after an expression, or on the last character of a variable. To simplify the display, a count is not shown if it is equal to the count of an earlier expression on the same line.
>
> The character '`=`' following the count for an expression says that the expression has returned the same value each time it was evaluated. In other words, it is not yet "covered" for coverage testing purposes.
>
> To clear the frequency count and coverage data for a definition, simply reinstrument it with `eval-defun`.

For example, after evaluating `(fac 5)` with a source breakpoint, and setting `edebug-test-coverage` to `t`, when the breakpoint is reached, the frequency data looks like this:

```
(defun fac (n)
  (if (= n 0) (edebug))
;#6           1      = =5
  (if (< 0 n)
;#5         =
      (* n (fac (1- n)))
;#    5               0
    1))
;#   0
```

The comment lines show that `fac` was called 6 times. The first `if` statement returned 5 times with the same result each time; the same is true of the condition on the second `if`. The recursive call of `fac` did not return at all.

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 increased 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 28.23 [Window Configurations], page 60, vol. 2). When you exit Edebug, it restores the previous window configuration.

Emacs redisplays only when it pauses. Usually, when you continue execution, the program re-enters Edebug at a breakpoint or after stepping, without pausing or reading input in between. In such cases, Emacs never gets a chance to redisplay the "outside" configuration. Consequently, what you see is the same window configuration as the last time Edebug was active, with no interruption.

Entry to Edebug for displaying something also saves and restores the following data (though some of them are deliberately not restored if an error or quit signal occurs).

- Which buffer is current, and the positions of point and the mark in the current buffer, are saved and restored.
- The outside window configuration is saved and restored if `edebug-save-windows` is non-`nil` (see Section 18.2.16 [Edebug Options], page 269).

  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 225, vol. 2.
- The variables `last-command`, `this-command`, `last-command-event`, `last-input-event`, `last-event-frame`, `last-nonmenu-event`, and `track-mouse`. Commands in Edebug do not affect these variables outside of Edebug.

  Executing commands within Edebug can change the key sequence that would be returned by `this-command-keys`, and there is no way to reset the key sequence from Lisp.

  Edebug cannot save and restore the value of `unread-command-events`. Entering Edebug while this variable has a nontrivial value can interfere with execution of the program you are debugging.
- Complex commands executed while in Edebug are added to the variable `command-history`. In rare cases this can alter execution.
- Within Edebug, the recursion depth appears one deeper than the recursion depth outside Edebug. This is not true of the automatically updated evaluation list window.
- `standard-output` and `standard-input` are bound to `nil` by the `recursive-edit`, but Edebug temporarily restores them during evaluations.
- The state of keyboard macro definition is saved and restored. While Edebug is active, `defining-kbd-macro` is bound to `edebug-continue-kbd-macro`.

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 13.5.2 [Argument Evaluation], page 185).

```
(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 183, for more explanation of the `declare` form.

Take care to ensure that the specifications are known to Edebug when you instrument code. If you are instrumenting a function from a file that uses `eval-when-compile` to require another file containing macro definitions, you may need to explicitly load that file.

You can also define an edebug specification for a macro separately from the macro definition with `def-edebug-spec`. Adding `debug` declarations is preferred, and more convenient, for macro definitions in Lisp, but `def-edebug-spec` makes it possible to define Edebug specifications for special forms implemented in C.

`def-edebug-spec` *macro specification* [Macro]

Specify which expressions of a call to macro *macro* are forms to be evaluated. *specification* should be the edebug specification. Neither argument is evaluated.

The *macro* argument can actually be any symbol, not just a macro name.

Here is a table of the possibilities for *specification* and how each directs processing of arguments.

`t`	All arguments are instrumented for evaluation.
`0`	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.

`edebug-eval-macro-args` [User Option]

This controls the way Edebug treats macro arguments with no explicit Edebug specification. If it is `nil` (the default), none of the arguments is instrumented for evaluation. Otherwise, all arguments are instrumented.

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. For more details, see Section 18.2.15.3 [Backtracking], page 268.

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 269, for the referenced examples):

`sexp` A single unevaluated Lisp object, which is not instrumented.

`form` A single evaluated expression, which is instrumented.

`place` A place to store a value, as in the Common Lisp `setf` construct.

`body` Short for `&rest form`. See `&rest` below.

`function-form`
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 `quote` rather than `function`, since it instruments the body of the lambda expression either way.

`lambda-expr`
A lambda expression with no quoting.

`&optional`
All following elements in the specification list are optional; as soon as one does not match, Edebug stops matching at this level.

To make just a few elements optional, followed by non-optional elements, use `[&optional` *specs*`...]`. To specify that several elements must all match or none, use `&optional [`*specs*`...]`. See the `defun` example.

`&rest` All following elements in the specification list are repeated zero or more times. In the last repetition, however, it is not a problem if the expression runs out before matching all of the elements of the specification list.

To repeat only a few elements, use `[&rest` *specs*`...]`. To specify several elements that must all match on every repetition, use `&rest [`*specs*`...]`.

`&or` Each of the following elements in the specification list is an alternative. One of the alternatives must match, or the `&or` specification fails.

Each list element following `&or` is a single alternative. To group two or more list elements as a single alternative, enclose them in `[...]`.

`&not` Each of the following elements is matched as alternatives as if by using `&or`, but if any of them match, the specification fails. If none of them match, nothing is matched, but the `&not` specification succeeds.

`&define` Indicates that the specification is for a defining form. The defining form itself is not instrumented (that is, Edebug does not stop before and after the defining form), but forms inside it typically will be instrumented. The `&define` keyword should be the first element in a list specification.

`nil` This is successful when there are no more arguments to match at the current argument list level; otherwise it fails. See sublist specifications and the backquote example.

`gate` No argument is matched but backtracking through the gate is disabled while matching the remainder of the specifications at this level. This is primarily used to generate more specific syntax error messages. See Section 18.2.15.3 [Backtracking], page 268, 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-edebug-spec` just as for macros. See the `defun` example.

  Otherwise, the symbol should be a predicate. The predicate is called with the argument, and if the predicate returns `nil`, the specification fails and the argument is not instrumented.

  Some suitable predicates include `symbolp`, `integerp`, `stringp`, `vectorp`, and `atom`.

`[`*`elements`*`...]`
: A vector of elements groups the elements into a single *group specification*. Its meaning has nothing to do with vectors.

`"`*`string`*`"` The argument should be a symbol named *string*. This specification is equivalent to the quoted symbol, `'`*`symbol`*, where the name of *symbol* is the *string*, but the string form is preferred.

`(vector` *`elements`*`...)`
: The argument should be a vector whose elements must match the *elements* in the specification. See the backquote example.

`(`*`elements`*`...)`
: Any other list is a *sublist specification* and the argument must be a list whose elements match the specification *elements*.

  A sublist specification may be a dotted list and the corresponding list argument may then be a dotted list. Alternatively, the last CDR of a dotted list specification may be another sublist specification (via a grouping or an indirect specification, e.g., `(spec . [(more specs...)])`) whose elements match the non-dotted list arguments. This is useful in recursive specifications such as in the backquote example. Also see the description of a `nil` specification above for terminating such recursion.

  Note that a sublist specification written as `(specs . nil)` is equivalent to `(specs)`, and `(specs . (sublist-elements...))` is equivalent to `(specs sublist-elements...)`.

Here is a list of additional specifications that may appear only after `&define`. See the `defun` example.

`name` 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.

`:name` This construct does not actually match an argument. The element following `:name` 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.

`arg` The argument, a symbol, is the name of an argument of the defining form. However, lambda-list keywords (symbols starting with '`&`') are not allowed.

`lambda-list`
This matches a lambda list—the argument list of a lambda expression.

`def-body` The argument is the body of code in a definition. This is like `body`, described above, but a definition body must be instrumented with a different Edebug call that looks up information associated with the definition. Use `def-body` for the highest level list of forms within the definition.

`def-form` The argument is a single, highest-level form in a definition. This is like `def-body`, except it is used to match a single form rather than a list of forms. As a special case, `def-form` also means that tracing information is not output when the form is executed. See the `interactive` example.

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; 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 the associated argument list and `interactive` specifications. It is necessary to handle interactive forms specially since an expression argument is actually evaluated outside of the function body. (The specification for `defmacro` is very similar to that for `defun`, but allows for the `declare` statement.)

```
(def-edebug-spec defun
  (&define name lambda-list
           [&optional stringp]   ; Match the doc string, if present.
           [&optional ("interactive" interactive)]
           def-body))

(def-edebug-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 is a little different, and does not support dotted lists because doing so causes very deep recursion that could fail.)

```
(def-edebug-spec \` (backquote-form))   ; Alias just for clarity.

(def-edebug-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 reset `edebug-setup-hook` to `nil`. You could use this to load up Edebug specifications associated with a package you are using, but only when you also use Edebug. See Section 18.2.2 [Instrumenting], page 253.

`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 253.

`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 253.

`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 263.

`edebug-save-displayed-buffer-points` [User Option]

If this is non-`nil`, Edebug saves and restores point in all displayed buffers.

Saving and restoring point in other buffers is necessary if you are debugging code that changes the point of a buffer that is displayed in a non-selected window. If Edebug or the user then selects the window, point in that buffer will move to the window's value of point.

Saving and restoring point in all buffers is expensive, since it requires selecting each window twice, so enable this only if you need it. See Section 18.2.14.2 [Edebug Display Update], page 263.

`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 253.

`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 261.

`edebug-test-coverage` [User Option]

If non-`nil`, Edebug tests coverage of all expressions debugged. See Section 18.2.13 [Coverage Testing], page 262.

`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 253.

`edebug-unwrap-results` [User Option]

If non-`nil`, Edebug tries to remove any of its own instrumentation when showing the results of expressions. This is relevant when debugging macros where the results of expressions are themselves instrumented expressions. As a very artificial example, suppose that the example function `fac` has been instrumented, and consider a macro of the form:

```
(defmacro test () "Edebug example."
  (if (symbol-function 'fac)
      ...))
```

If you instrument the `test` macro and step through it, then by default the result of the `symbol-function` call has numerous `edebug-after` and `edebug-before` forms, which can make it difficult to see the “actual” result. If `edebug-unwrap-results` is non-`nil`, Edebug tries to remove these forms from the result.

`edebug-on-error` [User Option]

Edebug binds `debug-on-error` to this value, if `debug-on-error` was previously `nil`. See Section 18.2.7 [Trapping Errors], page 258.

`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 258.

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 257.

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 262). These features partly duplicate each other, and it would be cleaner to combine them.

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 *buffer*, starting with the character directly after point. Point advances as characters are read.

marker The input characters are read from the buffer that *marker* is in, starting with the character directly after the marker. The marker position advances as characters are read. The value of point in the buffer has no effect when the stream is a marker.

string The input characters are taken from *string*, starting at the first character in the string and using as many characters as required.

function The input characters are generated by *function*, which must support two kinds of calls:

- When it is called with no arguments, it should return the next character.
- When it is called with one argument (always a character), *function* should save the argument and arrange to return it on the next call. This is called *unreading* the character; it happens when the Lisp reader reads one character too many and wants to "put it back where it came from". In this case, it makes no difference what value *function* returns.

`t` `t` used as a stream means that the input is read from the minibuffer. In fact, the minibuffer is invoked once and the text given by the user is made into a string that is then used as the input stream. If Emacs is running in batch mode, 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.

`nil` `nil` supplied as an input stream means to use the value of `standard-input` instead; that value is the *default input stream*, and must be a non-`nil` input stream.

symbol A symbol as input stream is equivalent to the symbol's function definition (if any).

Here is an example of reading from a stream that is a buffer, showing where point is located before and after:

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

(read (get-buffer "foo"))
     ⇒ is
(read (get-buffer "foo"))
     ⇒ the

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

Note that the first read skips a space. Reading skips any amount of whitespace preceding the significant text.

Here is an example of reading from a stream that is a marker, initially positioned at the beginning of the buffer shown. The value read is the symbol `This`.

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

```
(setq m (set-marker (make-marker) 1 (get-buffer "foo")))
     ⇒ #<marker at 1 in foo>
(read m)
     ⇒ This
m
     ⇒ #<marker at 5 in foo>   ;; Before the first space.
```

Here we read from the contents of a string:

```
(read "(When in) the course")
     ⇒ (When in)
```

The following example reads from the minibuffer. The prompt is: '`Lisp expression: `'. (That is always the prompt used when you read from the stream `t`.) The user's input is shown following the prompt.

```
(read t)
     ⇒ 23
---------- Buffer: Minibuffer ----------
Lisp expression: 23 RET
---------- Buffer: Minibuffer ----------
```

Finally, here is an example of a stream that is a function, named `useless-stream`. Before we use the stream, we initialize the variable `useless-list` to a list of characters. Then each call to the function `useless-stream` obtains the next character in the list or unreads a character by adding it to the front of the list.

```
(setq useless-list (append "XY()" nil))
     ⇒ (88 89 40 41)

(defun useless-stream (&optional unread)
  (if unread
      (setq useless-list (cons unread useless-list))
    (prog1 (car useless-list)
           (setq useless-list (cdr useless-list)))))
     ⇒ useless-stream
```

Now we read using the stream thus constructed:

```
(read 'useless-stream)
     ⇒ XY

useless-list
     ⇒ (40 41)
```

Note that the open and close parentheses remain in the list. The Lisp reader encountered the open parenthesis, decided that it ended the input, and unread it. Another attempt to read from the stream at this point would read '`()`' and return `nil`.

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.

`read-circle` [Variable]

If non-`nil`, this variable enables the reading of circular and shared structures. See Section 2.5 [Circular Objects], page 26. Its default value is `t`.

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.

marker The output characters are inserted into the buffer that *marker* points into, at the marker position. The marker position advances as characters are inserted. The value of point in the buffer has no effect on printing when the stream is a marker, and this kind of printing does not move point (except that if the marker points at or before the position of point, point advances with the surrounding text, as usual).

function The output characters are passed to *function*, which is responsible for storing them away. It is called with a single character as argument, as many times as there are characters to be output, and is responsible for storing the characters wherever you want to put them.

`t` The output characters are displayed in the echo area.

`nil` `nil` specified as an output stream means to use the value of `standard-output` instead; that value is the *default output stream*, and must not be `nil`.

symbol A symbol as output stream is equivalent to the symbol's function definition (if any).

Many of the valid output streams are also valid as input streams. The difference between input and output streams is therefore more a matter of how you use a Lisp object, than of different types of object.

Here is an example of a buffer used as an output stream. Point is initially located as shown immediately before the '`h`' in '`the`'. At the end, point is located directly before that same '`h`'.

```
---------- Buffer: foo ----------
This is t⋆he contents of foo.
---------- Buffer: foo ----------

(print "This is the output" (get-buffer "foo"))
     ⇒ "This is the output"

---------- Buffer: foo ----------
This is t
"This is the output"
⋆he contents of foo.
---------- Buffer: foo ----------
```

Now we show a use of a marker as an output stream. Initially, the marker is in buffer `foo`, between the '`t`' and the '`h`' in the word '`the`'. At the end, the marker has advanced over the inserted text so that it remains positioned before the same '`h`'. Note that the location of point, shown in the usual fashion, has no effect.

```
---------- Buffer: foo ----------
This is the ⋆output
---------- Buffer: foo ----------

(setq m (copy-marker 10))
     ⇒ #<marker at 10 in foo>
```

```
(print "More output for foo." m)
     ⇒ "More output for foo."

---------- Buffer: foo ----------
This is t
"More output for foo."
he ⋆output
---------- Buffer: foo ----------

m
     ⇒ #<marker at 34 in foo>
```

The following example shows output to the echo area:

```
(print "Echo Area output" t)
     ⇒ "Echo Area output"
---------- Echo Area ----------
"Echo Area output"
---------- Echo Area ----------
```

Finally, we show the use of a function as an output stream. The function `eat-output` takes each character that it is given and conses it onto the front of the list `last-output` (see Section 5.4 [Building Lists], page 68). 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 57, for other ways to obtain the printed representation of a Lisp object as a string.

`with-output-to-string` *body...* [Macro]

This macro executes the *body* forms with `standard-output` set up to feed output into a string. Then it returns that string.

For example, if the current buffer name is '`foo`',

```
(with-output-to-string
  (princ "The buffer is ")
  (princ (buffer-name)))
```

returns `"The buffer is foo"`.

`pp` *object* **&optional** *stream* [Function]

This function outputs *object* to *stream*, just like `prin1`, but does it in a more "pretty" way. That is, it'll indent and fill the object to make it more readable for humans.

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, e.g. `(quote foo)` prints as `'foo`, and `(function foo)` as `#'foo`.

`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. See Section 2.5 [Circular Objects], page 26.

`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. The default is `nil`, meaning use the shortest output that represents the number without losing information.

To control output format more precisely, you can put a string in this variable. The string should hold a '`%`'-specification to be used in the C function `sprintf`. For further restrictions on what you can use, see the variable's documentation string.

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 302, vol. 2), 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 83, vol. 2.)

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 32.19.9 [Fields], page 172, vol. 2), so that certain motion functions, including `beginning-of-line`, `forward-word`, `forward-sentence`, and `forward-paragraph`, stop at the boundary between the prompt and the actual text.

The minibuffer’s window is normally a single line; it grows automatically if the contents require more space. Whilst it is active, you can explicitly resize it temporarily with the window sizing commands; it reverts to its normal size when the minibuffer is exited. When the minibuffer is not active, you can resize it permanently by using the window sizing commands in the frame’s other window, or dragging the mode line with the mouse. (Due to details of the current implementation, for this to work `resize-mini-windows` must be `nil`.) If the frame contains just a minibuffer, 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 21.5 [Command Loop Info], page 324). Your program should bind them around the code that uses the minibuffer, if you do not want that to change them.

Under some circumstances, a command can use a minibuffer even if there is an active minibuffer; such a minibuffer is called a *recursive minibuffer*. The first minibuffer is named ‘ `*Minibuf-1*`’. Recursive minibuffers are named by incrementing the number at the end of the name. (The names begin with a space so that they won’t show up in normal buffer lists.) Of several recursive minibuffers, the innermost (or most recently entered) is the active minibuffer. 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 313.)

Like other buffers, a minibuffer uses a local keymap (see Chapter 22 [Keymaps], page 360) 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 285, for the non-completion minibuffer local maps. See Section 20.6.3 [Completion Commands], page 296, for the minibuffer local maps for completion.

When a minibuffer is inactive, its major mode is `minibuffer-inactive-mode`, with keymap `minibuffer-inactive-mode-map`. This is only really useful if the minibuffer is in a separate frame. See Section 29.8 [Minibuffers and Frames], page 83, vol. 2.

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

The most basic primitive for minibuffer input is `read-from-minibuffer`, which can be used to read either a string or a Lisp object in textual form. The function `read-regexp` is used for reading regular expressions (see Section 34.3 [Regular Expressions], page 211, vol. 2), which are a special kind of string. There are also specialized functions for reading commands, variables, file names, etc. (see Section 20.6 [Completion], page 291).

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 316.

`read-from-minibuffer` *prompt* **&optional** *initial keymap read history default inherit-input-method* [Function]

This function is the most general way to get input from the minibuffer. By default, it accepts arbitrary text and returns it as a string; however, if *read* is non-`nil`, then it uses `read` to convert the text into a Lisp object (see Section 19.3 [Input Functions], page 276).

The first thing this function does is to activate a minibuffer and display it with *prompt* (which must be a string) as the prompt. Then the user can edit text in the minibuffer.

When the user types a command to exit the minibuffer, `read-from-minibuffer` constructs the return value from the text in the minibuffer. Normally it returns a string containing that text. However, if *read* is non-`nil`, `read-from-minibuffer` reads the text and returns the resulting Lisp object, unevaluated. (See Section 19.3 [Input Functions], page 276, for information about reading.)

The argument *default* specifies default values to make available through the history commands. It should be a string, a list of strings, or `nil`. The string or strings become the minibuffer's "future history", available to the user with *M-n*.

If *read* is non-`nil`, then *default* is also used as the input to `read`, if the user enters empty input. If *default* is a list of strings, the first string is used as the input. If *default* is `nil`, empty input results in an `end-of-file` error. However, in the usual case (where *read* is `nil`), `read-from-minibuffer` ignores *default* when the user enters empty input and returns an empty string, `""`. In this respect, it differs from all the other minibuffer input functions in this chapter.

If *keymap* is non-`nil`, that keymap is the local keymap to use in the minibuffer. If *keymap* is omitted or `nil`, the value of `minibuffer-local-map` is used as the keymap. Specifying a keymap is the most important way to customize the minibuffer for various applications such as completion.

The argument *history* specifies a history list variable to use for saving the input and for history commands used in the minibuffer. It defaults to `minibuffer-history`. You can optionally specify a starting position in the history list as well. See Section 20.4 [Minibuffer History], page 289.

If the variable `minibuffer-allow-text-properties` is non-`nil`, then the string that is returned includes whatever text properties were present in the minibuffer. Otherwise all the text properties are stripped when the value is returned.

If the argument *inherit-input-method* is non-`nil`, then the minibuffer inherits the current input method (see Section 33.10 [Input Methods], page 206, vol. 2) and the setting of `enable-multibyte-characters` (see Section 33.1 [Text Representations], page 182, vol. 2) from whichever buffer was current before entering the minibuffer.

Use of *initial* is mostly deprecated; we recommend using a non-`nil` value only in conjunction with specifying a cons cell for *history*. See Section 20.5 [Initial Input], page 291.

`read-string` *prompt* **&optional** *initial history default inherit-input-method* [Function]

This function reads a string from the minibuffer and returns it. The arguments *prompt*, *initial*, *history* and *inherit-input-method* are used as in `read-from-minibuffer`. The keymap used is `minibuffer-local-map`.

The optional argument *default* is used as in `read-from-minibuffer`, except that, if non-`nil`, it also specifies a default value to return if the user enters null input. As in `read-from-minibuffer` it should be a string, a list of strings, or `nil`, which is equivalent to an empty string. When *default* is a string, that string is the default value. When it is a list of strings, the first string is the default value. (All these strings are available to the user in the “future minibuffer history”.)

This function works by calling the `read-from-minibuffer` function:

```
(read-string prompt initial history default inherit)
≡
(let ((value
       (read-from-minibuffer prompt initial nil nil
                             history default inherit)))
  (if (and (equal value "") default)
      (if (consp default) (car default) default)
    value))
```

`read-regexp` *prompt* **&optional** *default* [Function]

This function reads a regular expression as a string from the minibuffer and returns it. The argument *prompt* is used as in `read-from-minibuffer`. The keymap used is `minibuffer-local-map`, and `regexp-history` is used as the history list (see Section 20.4 [Minibuffer History], page 289).

The optional argument *default* specifies a default value to return if the user enters null input; it should be a string, or `nil`, which is equivalent to an empty string.

In addition, `read-regexp` collects a few useful candidates for input and passes them to `read-from-minibuffer`, to make them available to the user as the “future minibuffer history list” (see Section “Minibuffer History” in *The GNU Emacs Manual*). These candidates are:

- The word or symbol at point.
- The last regexp used in an incremental search.
- The last string used in an incremental search.
- The last string or pattern used in query-replace commands.

This function works by calling the `read-from-minibuffer` function, after computing the list of defaults as described above.

`minibuffer-allow-text-properties` [Variable]

If this variable is `nil`, then `read-from-minibuffer` and `read-string` strip all text properties from the minibuffer input before returning it. However, `read-no-blanks-input` (see below), as well as `read-minibuffer` and related functions (see Section 20.3 [Reading Lisp Objects With the Minibuffer], page 288), 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:

C-j	`exit-minibuffer`
RET	`exit-minibuffer`
C-g	`abort-recursive-edit`
M-n DOWN	`next-history-element`
M-p UP	`previous-history-element`
M-s	`next-matching-history-element`
M-r	`previous-matching-history-element`

`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, evaluates it, then returns the result. The difference between this command and `eval-minibuffer` is that here the initial *form* is not optional and it is treated as a Lisp object to be converted to printed representation rather than as a string of text. It is printed with `prin1`, so if it is a string, double-quote characters (‘`"`’) appear in the initial text. See Section 19.5 [Output Functions], page 279.

In the following example, we offer the user an expression with initial text that is already a valid form:

```
(edit-and-eval-command "Please edit: " '(forward-word 1))

;; After evaluation of the preceding expression,
;;     the following appears in the minibuffer:

---------- Buffer: Minibuffer ----------
Please edit: (forward-word 1)⋆
---------- Buffer: Minibuffer ----------
```

Typing RET right away would exit the minibuffer and evaluate the expression, thus moving point forward one word.

20.4 Minibuffer History

A *minibuffer history list* records previous minibuffer inputs so the user can reuse them conveniently. It is a variable whose value is a list of strings (previous inputs), most recent first.

There are many separate minibuffer history lists, used for different kinds of inputs. It's the Lisp programmer's job to specify the right history list for each use of the minibuffer.

You specify a minibuffer history list with the optional *history* argument to `read-from-minibuffer` or `completing-read`. Here are the possible values for it:

variable Use *variable* (a symbol) as the history list.

(*variable* . *startpos*)
: Use *variable* (a symbol) as the history list, and assume that the initial history position is *startpos* (a nonnegative integer).

  Specifying 0 for *startpos* is equivalent to just specifying the symbol *variable*. `previous-history-element` will display the most recent element of the history list in the minibuffer. If you specify a positive *startpos*, the minibuffer history functions behave as if `(elt` ***`variable`*** `(1-` ***`startpos`***`))` were the history element currently shown in the minibuffer.

  For consistency, you should also specify that element of the history as the initial minibuffer contents, using the *initial* argument to the minibuffer input function (see Section 20.5 [Initial Input], page 291).

If you don't specify *history*, then the default history list `minibuffer-history` is used. For other standard history lists, see below. You can also create your own history list variable; just initialize it to `nil` before the first use.

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`. The default value is `t`.

`history-length` [User Option]

The value of this variable specifies the maximum length for all history lists that don't specify their own maximum lengths. If the value is `t`, that means there is no maximum (don't delete old elements). If a history list variable's symbol has a non-`nil` `history-length` property, it overrides this variable for that particular history list.

`history-delete-duplicates` [User Option]

If the value of this variable is `t`, that means when adding a new history element, all previous identical elements are deleted.

Here are some of the standard minibuffer history list variables:

`minibuffer-history` [Variable]

The default history list for minibuffer history input.

`query-replace-history` [Variable]

A history list for arguments to `query-replace` (and similar arguments to other commands).

`file-name-history` [Variable]

A history list for file-name arguments.

`buffer-name-history` [Variable]

A history list for buffer-name arguments.

`regexp-history` [Variable]

A history list for regular expression arguments.

`extended-command-history` [Variable]

A history list for arguments that are names of extended commands.

`shell-command-history` [Variable]

A history list for arguments that are shell commands.

`read-expression-history` [Variable]

A history list for arguments that are Lisp expressions to evaluate.

`face-name-history` [Variable]

A history list for arguments that are faces.

20.5 Initial Input

Several of the functions for minibuffer input have an argument called *initial*. This is a mostly-deprecated feature for specifying that the minibuffer should start out with certain text, instead of empty as usual.

If *initial* is a string, the minibuffer starts out containing the text of the string, with point at the end, when the user starts to edit the text. If the user simply types `RET` to exit the minibuffer, it will use the initial input string to determine the value to return.

We discourage use of a non-`nil` value for *initial*, because initial input is an intrusive interface. History lists and default values provide a much more convenient method to offer useful default inputs to the user.

There is just one situation where you should specify a string for an *initial* argument. This is when you specify a cons cell for the *history* argument. See Section 20.4 [Minibuffer History], page 289.

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.

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 following completion functions have nothing in themselves to do with minibuffers. We describe them here to keep them near the higher-level completion features that do use the minibuffer.

`try-completion` *string collection* **&optional** *predicate* [Function]

This function returns the longest common substring of all possible completions of *string* in *collection*.

The *collection* argument is called the *completion table*. Its value must be a list of strings, an alist whose keys are strings or symbols, an obarray, a hash table, or a completion function.

Completion compares *string* against each of the permissible completions specified by *collection*. If no permissible completions match, `try-completion` returns `nil`. If there is just one matching completion, and the match is exact, it returns `t`. Otherwise, it returns the longest initial sequence common to all possible matching completions.

If *collection* is an alist (see Section 5.8 [Association Lists], page 82), the permissible completions are the elements of the alist that are either strings, 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 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.

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 function as *collection*. Then the function is solely responsible for performing completion; `try-completion` returns whatever this function returns. The function is called with three arguments: *string*, *predicate* and `nil` (the third argument is so that the same function can be used in `all-completions` and do the appropriate thing in either case). See Section 20.6.7 [Programmed Completion], page 305.

If the argument *predicate* is non-`nil`, then it must be a function of one argument, unless *collection* is a hash table, in which case it should be a function of two arguments. It is used to test each possible match, and the match is accepted only if *predicate* returns non-`nil`. The argument given to *predicate* is either a string or a cons cell (the CAR of which is a string) from the alist, or a symbol (*not* a symbol name) from the obarray. If *collection* is a hash table, *predicate* is called with two arguments, the string key and the associated value.

In addition, to be acceptable, a completion must also match all the regular expressions in `completion-regexp-list`. (Unless *collection* is a function, in which case that function has to handle `completion-regexp-list` itself.)

In the first of the following examples, the string '`foo`' is matched by three of the alist CARs. All of the matches begin with the characters '`fooba`', so that is the result. In the second example, there is only one possible match, and it is exact, so the return value is `t`.

```
(try-completion
 "foo"
 '(("foobar1" 1) ("barfoo" 2) ("foobaz" 3) ("foobar2" 4)))
     ⇒ "fooba"

(try-completion "foo" '(("barfoo" 2) ("foo" 3)))
     ⇒ t
```

In the following example, numerous symbols begin with the characters '`forw`', and all of them begin with the word '`forward`'. In most of the symbols, this is followed with a '`-`', but not in all, so no more than '`forward`' can be completed.

```
(try-completion "forw" obarray)
     ⇒ "forward"
```

Finally, in the following example, only two of the three possible matches pass the predicate `test` (the string '`foobaz`' is too short). Both of those begin with the string '`foobar`'.

```
(defun test (s)
  (> (length (car s)) 6))
     ⇒ test
(try-completion
 "foo"
 '(("foobar1" 1) ("barfoo" 2) ("foobaz" 3) ("foobar2" 4))
 'test)
     ⇒ "foobar"
```

`all-completions` *string collection* **&optional** *predicate* [Function]

This function returns a list of all possible completions of *string*. The arguments to this function are the same as those of `try-completion`, and it uses `completion-regexp-list` in the same way that `try-completion` does.

If *collection* is a function, it is called with three arguments: *string*, *predicate* and `t`; then `all-completions` returns whatever the function returns. See Section 20.6.7 [Programmed Completion], page 305.

Here is an example, using the function `test` shown in the example for `try-completion`:

```
(defun test (s)
  (> (length (car s)) 6))
     ⇒ test

(all-completions
 "foo"
 '(("foobar1" 1) ("barfoo" 2) ("foobaz" 3) ("foobar2" 4))
 'test)
     ⇒ ("foobar1" "foobar2")
```

`test-completion` *string collection* **&optional** *predicate* [Function]

This function returns non-`nil` if *string* is a valid completion alternative specified by *collection* and *predicate*. The arguments are the same as in `try-completion`. For instance, if *collection* is a list of strings, this is true if *string* appears in the list and *predicate* is satisfied.

This function uses `completion-regexp-list` in the same way that `try-completion` does.

If *predicate* is non-`nil` and if *collection* contains several strings that are equal to each other, as determined by `compare-strings` according to `completion-ignore-case`, then *predicate* should accept either all or none of them. Otherwise, the return value of `test-completion` is essentially unpredictable.

If *collection* is a function, it is called with three arguments, the values *string*, *predicate* and `lambda`; whatever it returns, `test-completion` returns in turn.

`completion-boundaries` *string collection predicate suffix* [Function]

This function returns the boundaries of the field on which *collection* will operate, assuming that *string* holds the text before point and *suffix* holds the text after point.

Normally completion operates on the whole string, so for all normal collections, this will always return `(0 . (length suffix))`. But more complex completion such as completion on files is done one field at a time. For example, completion of `"/usr/sh"` will include `"/usr/share/"` but not `"/usr/share/doc"` even if `"/usr/share/doc"` exists. Also `all-completions` on `"/usr/sh"` will not include `"/usr/share/"` but only `"share/"`. So if *string* is `"/usr/sh"` and *suffix* is `"e/doc"`, `completion-boundaries` will return `(5 . 1)` which tells us that the *collection* will only return completion information that pertains to the area after `"/usr/"` and before `"/doc"`.

If you store a completion alist in a variable, you should mark the variable as "risky" by giving it a non-`nil` `risky-local-variable` property. See Section 11.11 [File Local Variables], page 156.

`completion-ignore-case` [Variable]

If the value of this variable is non-`nil`, case is not considered significant in completion. Within `read-file-name`, this variable is overridden by `read-file-name-completion-ignore-case` (see Section 20.6.5 [Reading File Names], page 300); within `read-buffer`, it is overridden by `read-buffer-completion-ignore-case` (see Section 20.6.4 [High-Level Completion], page 298).

`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 211, vol. 2) 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:

```
(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 initial history default inherit-input-method* [Function]

This function reads a string in the minibuffer, assisting the user by providing completion. It activates the minibuffer with prompt *prompt*, which must be a string.

The actual completion is done by passing the completion table *collection* and the completion predicate *predicate* to the function `try-completion` (see Section 20.6.1

[Basic Completion], page 291). 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 293.

The value of the optional argument *require-match* determines how the user may exit the minibuffer:

- If `nil`, the usual minibuffer exit commands work regardless of the input in the minibuffer.
- If `t`, the usual minibuffer exit commands won't exit unless the input completes to an element of *collection*.
- If `confirm`, the user can exit with any input, but is asked for confirmation if the input is not an element of *collection*.
- If `confirm-after-completion`, the user can exit with any input, but is asked for confirmation if the preceding command was a completion command (i.e., one of the commands in `minibuffer-confirm-exit-commands`) and the resulting input is not an element of *collection*. See Section 20.6.3 [Completion Commands], page 296.
- Any other value of *require-match* behaves like `t`, except that the exit commands won't exit if it performs completion.

However, empty input is always permitted, regardless of the value of *require-match*; in that case, `completing-read` returns the first element of *default*, if it is a list; `""`, if *default* is `nil`; or *default*. The string or strings in *default* are also available to the user through the history commands.

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 296.

The argument *history* 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 289.

The argument *initial* is mostly deprecated; we recommend using a non-`nil` value only in conjunction with specifying a cons cell for *history*. See Section 20.5 [Initial Input], page 291. 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 33.10 [Input Methods], page 206, vol. 2) and the setting of `enable-multibyte-characters` (see Section 33.1 [Text Representations], page 182, vol. 2) from whichever buffer was current before entering the minibuffer.

If the variable `completion-ignore-case` is non-`nil`, completion ignores case when comparing the input against the possible matches. See Section 20.6.1 [Basic Completion], page 291. In this mode of operation, *predicate* must also ignore case, or you will get surprising results.

Here's an example of using `completing-read`:

```
(completing-read
 "Complete a foo: "
 '(("foobar1" 1) ("barfoo" 2) ("foobaz" 3) ("foobar2" 4))
 nil t "fo")
```

```
;; After evaluation of the preceding expression,
;;    the following appears in the minibuffer:

---------- Buffer: Minibuffer ----------
Complete a foo: fo⋆
---------- Buffer: Minibuffer ----------
```

If the user then types *DEL DEL b RET*, `completing-read` returns `barfoo`.

The `completing-read` function binds variables to pass information to the commands that actually do completion. They are described in the following section.

`completing-read-function` [Variable]

The value of this variable must be a function, which is called by `completing-read` to actually do its work. It should accept the same arguments as `completing-read`. This can be bound to a different function to completely override the normal behavior of `completing-read`.

20.6.3 Minibuffer Commands that Do Completion

This section describes the keymaps, commands and user options used in the minibuffer to do completion.

`minibuffer-completion-table` [Variable]

The value of this variable is the completion table 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]

This variable determines whether Emacs asks for confirmation before exiting the minibuffer; `completing-read` binds this variable, and the function `minibuffer-complete-and-exit` checks the value before exiting. If the value is `nil`, confirmation is not required. If the value is `confirm`, the user may exit with an input that is not a valid completion alternative, but Emacs asks for confirmation. If the value is `confirm-after-completion`, the user may exit with an input that is not a valid completion alternative, but Emacs asks for confirmation if the user submitted the input right after any of the completion commands in `minibuffer-confirm-exit-commands`.

`minibuffer-confirm-exit-commands` [Variable]

This variable holds a list of commands that cause Emacs to ask for confirmation before exiting the minibuffer, if the *require-match* argument to `completing-read` is `confirm-after-completion`. The confirmation is requested if the user attempts to exit the minibuffer immediately after calling any command in this list.

`minibuffer-complete-word` [Command]

This function completes the minibuffer contents by at most a single word. Even if the minibuffer contents have only one completion, `minibuffer-complete-word` does not add any characters beyond the first character that is not a word constituent. See Chapter 35 [Syntax Tables], page 234, vol. 2.

`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 274, 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`. A common way to use it is together with `with-output-to-temp-buffer`, like this:

```
(with-output-to-temp-buffer "*Completions*"
  (display-completion-list
    (all-completions (buffer-string) my-alist)
    (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:
>
> | | |
> |---|---|
> | `?` | `minibuffer-completion-help` |
> | `SPC` | `minibuffer-complete-word` |
> | `TAB` | `minibuffer-complete` |
>
> and uses `minibuffer-local-map` as its parent keymap (see [Definition of minibuffer-local-map], page 287).

`minibuffer-local-must-match-map` [Variable]

> `completing-read` uses this value as the local keymap when an exact match of one of the completions is required. Therefore, no keys are bound to `exit-minibuffer`, the command that exits the minibuffer unconditionally. By default, this keymap makes the following bindings:
>
> | | |
> |---|---|
> | *`C-j`* | `minibuffer-complete-and-exit` |
> | `RET` | `minibuffer-complete-and-exit` |
>
> and uses `minibuffer-local-completion-map` as its parent keymap.

`minibuffer-local-filename-completion-map` [Variable]

> This is a sparse keymap that simply unbinds `SPC`; because filenames can contain spaces. The function `read-file-name` combines this keymap with either `minibuffer-local-completion-map` or `minibuffer-local-must-match-map`.

20.6.4 High-Level Completion Functions

This section describes the higher-level convenience functions for reading certain sorts of names with completion.

In most cases, you should not call these functions in the middle of a Lisp function. When possible, do all minibuffer input as part of reading the arguments for a command, in the `interactive` specification. See Section 21.2 [Defining Commands], page 316.

`read-buffer` *prompt* **&optional** *default require-match* [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, a list of strings, or a buffer. If it is a list, the default value is the first element of this list. It is mentioned in the prompt, but is not inserted in the minibuffer as initial input.
>
> The argument *prompt* should be a string ending with a colon and a space. If *default* is non-`nil`, the function inserts it in *prompt* before the colon to follow the convention for reading from the minibuffer with a default value (see Section D.3 [Programming Tips], page 447, vol. 2).
>
> The optional argument *require-match* has the same meaning as in `completing-read`. See Section 20.6.2 [Minibuffer Completion], page 294.

In the following example, the user enters '`minibuffer.t`', and then types `RET`. The argument *require-match* is `t`, and the only buffer name starting with the given input is '`minibuffer.texi`', so that name is the value.

```
(read-buffer "Buffer name: " "foo" t)
;; After evaluation of the preceding expression,
;;     the following prompt appears,
;;     with an empty minibuffer:

---------- Buffer: Minibuffer ----------
Buffer name (default foo): ⋆
---------- Buffer: Minibuffer ----------

;; The user types minibuffer.t RET.
     ⇒ "minibuffer.texi"
```

`read-buffer-function` [User Option]

This variable specifies how to read buffer names. The function is called with the arguments passed to `read-buffer`. 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-buffer-completion-ignore-case` [User Option]

If this variable is non-`nil`, `read-buffer` ignores case when performing completion.

`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 321.

The argument *default* specifies what to return if the user enters null input. It can be a symbol, a string or a list of strings. If it is a string, `read-command` interns it before returning it. If it is a list, `read-command` interns the first element of this list. If *default* is `nil`, that means no default has been specified; then if the user enters null input, the return value is `(intern "")`, that is, a symbol whose name is an empty string.

```
(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. Its arguments have the same form as those of `read-command`. In general, this function is similar to `read-command`, but uses the predicate `user-variable-p` instead of `commandp`.

`read-color` **&optional** *prompt convert allow-empty display* [Command]

This function reads a string that is a color specification, either the color's name or an RGB hex value such as `#RRRGGGBBB`. It prompts with *prompt* (default: `"Color (name or #RGB triplet):"`) and provides completion for color names, but not for hex RGB values. In addition to names of standard colors, completion candidates include the foreground and background colors at point.

Valid RGB values are described in Section 29.20 [Color Names], page 92, vol. 2.

The function's return value is the string typed by the user in the minibuffer. However, when called interactively or if the optional argument *convert* is non-`nil`, it converts any input color name into the corresponding RGB value string and instead returns that. This function requires a valid color specification to be input. Empty color names are allowed when *allow-empty* is non-`nil` and the user enters null input.

Interactively, or when *display* is non-`nil`, the return value is also displayed in the echo area.

See also the functions `read-coding-system` and `read-non-nil-coding-system`, in Section 33.9.4 [User-Chosen Coding Systems], page 198, vol. 2, and `read-input-method-name`, in Section 33.10 [Input Methods], page 206, vol. 2.

20.6.5 Reading File Names

The high-level completion functions `read-file-name`, `read-directory-name`, and `read-shell-command` are designed to read file names, directory names, and shell commands, respectively. They provide special features, including automatic insertion of the default directory.

`read-file-name` *prompt* **&optional** *directory default require-match initial predicate* [Function]

This function reads a file name, prompting with *prompt* and providing completion.

As an exception, this function reads a file name using a graphical file dialog instead of the minibuffer, if all of the following are true:

1. It is invoked via a mouse command.
2. The selected frame is on a graphical display supporting such dialogs.
3. The variable `use-dialog-box` is non-`nil`. See Section "Dialog Boxes" in *The GNU Emacs Manual*.
4. The *directory* argument, described below, does not specify a remote file. See Section "Remote Files" in *The GNU Emacs Manual*.

The exact behavior when using a graphical file dialog is platform-dependent. Here, we simply document the behavior when using the minibuffer.

`read-file-name` does not automatically expand the returned file name. You must call `expand-file-name` yourself if an absolute file name is required.

The optional argument *require-match* has the same meaning as in `completing-read`. See Section 20.6.2 [Minibuffer Completion], page 294.

The argument *directory* specifies the directory to use for completing relative file names. It should be an absolute directory name. If the variable `insert-default-directory` is non-`nil`, *directory* is also inserted in the minibuffer as initial input. It defaults to the current buffer's value of `default-directory`.

If you specify *initial*, that is an initial file name to insert in the buffer (after *directory*, if that is inserted). In this case, point goes at the beginning of *initial*. The default for *initial* is `nil`—don't insert any file name. To see what *initial* does, try the command *C-x C-v* in a buffer visiting a file. **Please note:** we recommend using *default* rather than *initial* in most cases.

If *default* is non-`nil`, then the function returns *default* if the user exits the minibuffer with the same non-empty contents that `read-file-name` inserted initially. The initial minibuffer contents are always non-empty if `insert-default-directory` is non-`nil`, as it is by default. *default* is not checked for validity, regardless of the value of *require-match*. However, if *require-match* is non-`nil`, the initial minibuffer contents should be a valid file (or directory) name. Otherwise `read-file-name` attempts completion if the user exits without any editing, and does not return *default*. *default* is also available through the history commands.

If *default* is `nil`, `read-file-name` tries to find a substitute default to use in its place, which it treats in exactly the same way as if it had been specified explicitly. If *default* is `nil`, but *initial* is non-`nil`, then the default is the absolute file name obtained from *directory* and *initial*. If both *default* and *initial* are `nil` and the buffer is visiting a file, `read-file-name` uses the absolute file name of that file as default. If the buffer is not visiting a file, then there is no default. In that case, if the user types `RET` without any editing, `read-file-name` simply returns the pre-inserted contents of the minibuffer.

If the user types `RET` in an empty minibuffer, this function returns an empty string, regardless of the value of *require-match*. This is, for instance, how the user can make the current buffer visit no file using *M-x set-visited-file-name*.

If *predicate* is non-`nil`, it specifies a function of one argument that decides which file names are acceptable completion alternatives. A file name is an acceptable value if *predicate* returns non-`nil` for it.

Here is an example of using `read-file-name`:

```
(read-file-name "The file is ")

;; After evaluation of the preceding expression,
;;      the following appears in the minibuffer:
```

```
---------- Buffer: Minibuffer ----------
The file is /gp/gnu/elisp/⋆
---------- Buffer: Minibuffer ----------
```

Typing *manual TAB* results in the following:

```
---------- Buffer: Minibuffer ----------
The file is /gp/gnu/elisp/manual.texi⋆
---------- Buffer: Minibuffer ----------
```

If the user types `RET`, `read-file-name` returns the file name as the string `"/gp/gnu/elisp/manual.texi"`.

`read-file-name-function` [Variable]

If non-`nil`, this should be a function that accepts the same arguments as `read-file-name`. When `read-file-name` is called, it calls this function with the supplied arguments instead of doing its usual work.

`read-file-name-completion-ignore-case` [User Option]

If this variable is non-`nil`, `read-file-name` ignores case when performing completion.

`read-directory-name` *prompt* **&optional** *directory default require-match initial* [Function]

This function is like `read-file-name` but allows only directory names as completion alternatives.

If *default* is `nil` and *initial* is non-`nil`, `read-directory-name` constructs a substitute default by combining *directory* (or the current buffer's default directory if *directory* is `nil`) and *initial*. If both *default* and *initial* are `nil`, this function uses *directory* as substitute default, or the current buffer's default directory if *directory* is `nil`.

`insert-default-directory` [User Option]

This variable is used by `read-file-name`, and thus, indirectly, by most commands reading file names. (This includes all commands that use the code letters '`f`' or '`F`' in their interactive form. See Section 21.2.2 [Code Characters for interactive], page 318.) Its value controls whether `read-file-name` starts by placing the name of the default directory in the minibuffer, plus the initial file name, if any. If the value of this variable is `nil`, then `read-file-name` does not place any initial input in the minibuffer (unless you specify initial input with the *initial* argument). In that case, the default directory is still used for completion of relative file names, but is not displayed.

If this variable is `nil` and the initial minibuffer contents are empty, the user may have to explicitly fetch the next history element to access a default value. If the variable is non-`nil`, the initial minibuffer contents are always non-empty and the user can always request a default value by immediately typing `RET` in an unedited minibuffer. (See above.)

For example:

```
;; Here the minibuffer starts out with the default directory.
(let ((insert-default-directory t))
  (read-file-name "The file is "))
```

```
---------- Buffer: Minibuffer ----------
The file is ~lewis/manual/⋆
---------- Buffer: Minibuffer ----------

;; Here the minibuffer is empty and only the prompt
;;    appears on its line.
(let ((insert-default-directory nil))
  (read-file-name "The file is "))

---------- Buffer: Minibuffer ----------
The file is ⋆
---------- Buffer: Minibuffer ----------
```

`read-shell-command` *prompt* **&optional** *initial history* **&rest** *args* [Function]

This function reads a shell command from the minibuffer, prompting with *prompt* and providing intelligent completion. It completes the first word of the command using candidates that are appropriate for command names, and the rest of the command words as file names.

This function uses `minibuffer-local-shell-command-map` as the keymap for minibuffer input. The *history* argument specifies the history list to use; if is omitted or `nil`, it defaults to `shell-command-history` (see Section 20.4 [Minibuffer History], page 289). The optional argument *initial* specifies the initial content of the minibuffer (see Section 20.5 [Initial Input], page 291). The rest of *args*, if present, are used as the *default* and *inherit-input-method* arguments in `read-from-minibuffer` (see Section 20.2 [Text from Minibuffer], page 285).

`minibuffer-local-shell-command-map` [Variable]

This keymap is used by `read-shell-command` for completing command and file names that are part of a shell command. It uses `minibuffer-local-map` as its parent keymap, and binds `TAB` to `completion-at-point`.

20.6.6 Completion Variables

Here are some variables that can be used to alter the default completion behavior.

`completion-styles` [User Option]

The value of this variable is a list of completion style (symbols) to use for performing completion. A *completion style* is a set of rules for generating completions. Each symbol occurring this list must have a corresponding entry in `completion-styles-alist`.

`completion-styles-alist` [Variable]

This variable stores a list of available completion styles. Each element in the list has the form

```
(style try-completion all-completions doc)
```

Here, *style* is the name of the completion style (a symbol), which may be used in the `completion-styles` variable to refer to this style; *try-completion* is the function that does the completion; *all-completions* is the function that lists the completions; and *doc* is a string describing the completion style.

The *try-completion* and *all-completions* functions should each accept four arguments: *string*, *collection*, *predicate*, and *point*. The *string*, *collection*, and *predicate* arguments have the same meanings as in `try-completion` (see Section 20.6.1 [Basic Completion], page 291), and the *point* argument is the position of point within *string*. Each function should return a non-`nil` value if it performed its job, and `nil` if it did not (e.g. if there is no way to complete *string* according to the completion style).

When the user calls a completion command like `minibuffer-complete` (see Section 20.6.3 [Completion Commands], page 296), Emacs looks for the first style listed in `completion-styles` and calls its *try-completion* function. If this function returns `nil`, Emacs moves to the next listed completion style and calls its *try-completion* function, and so on until one of the *try-completion* functions successfully performs completion and returns a non-`nil` value. A similar procedure is used for listing completions, via the *all-completions* functions.

See Section "Completion Styles" in *The GNU Emacs Manual*, for a description of the available completion styles.

`completion-category-overrides` [User Option]

This variable specifies special completion styles and other completion behaviors to use when completing certain types of text. Its value should be an alist with elements of the form `(`*`category`*` . `*`alist`*`)`. *category* is a symbol describing what is being completed; currently, the `buffer`, `file`, and `unicode-name` categories are defined, but others can be defined via specialized completion functions (see Section 20.6.7 [Programmed Completion], page 305). *alist* is an association list describing how completion should behave for the corresponding category. The following alist keys are supported:

`styles` The value should be a list of completion styles (symbols).

`cycle` The value should be a value for `completion-cycle-threshold` (see Section "Completion Options" in *The GNU Emacs Manual*) for this category.

Additional alist entries may be defined in the future.

`completion-extra-properties` [Variable]

This variable is used to specify extra properties of the current completion command. It is intended to be let-bound by specialized completion commands. Its value should be a list of property and value pairs. The following properties are supported:

`:annotation-function`

The value should be a function to add annotations in the completions buffer. This function must accept one argument, a completion, and should either return `nil` or a string to be displayed next to the completion.

`:exit-function`

The value should be a function to run after performing completion. The function should accept two arguments, *string* and *status*, where *string* is the text to which the field was completed, and *status* indicates what kind of operation happened: `finished` if text is now complete, `sole` if the text cannot be further completed but completion is not finished, or `exact` if the text is a valid completion but may be further completed.

20.6.7 Programmed Completion

Sometimes it is not possible or convenient to create an alist or an obarray containing all the intended possible completions ahead of time. In such a case, you can supply your own function to compute the completion of a given string. This is called *programmed completion*. Emacs uses programmed completion when completing file names (see Section 25.8.6 [File Name Completion], page 489), among many other cases.

To use this feature, pass a function as the *collection* argument to `completing-read`. The function `completing-read` arranges to pass your completion function along to `try-completion`, `all-completions`, and other basic completion functions, which will then let your function do all the work.

The completion function should accept three arguments:

- The string to be completed.
- A predicate function with which to filter possible matches, or `nil` if none. The function should call the predicate for each possible match, and ignore the match if the predicate returns `nil`.
- A flag specifying the type of completion operation to perform. This is one of the following four values:

  `nil` This specifies a `try-completion` operation. The function should return `t` if the specified string is a unique and exact match; if there is more than one match, it should return the common substring of all matches (if the string is an exact match for one completion alternative but also matches other longer alternatives, the return value is the string); if there are no matches, it should return `nil`.

  `t` This specifies an `all-completions` operation. The function should return a list of all possible completions of the specified string.

  `lambda` This specifies a `test-completion` operation. The function should return `t` if the specified string is an exact match for some completion alternative; `nil` otherwise.

  `(boundaries . *suffix*)`
  This specifies a `completion-boundaries` operation. The function should return `(boundaries *start* . *end*)`, where *start* is the position of the beginning boundary in the specified string, and *end* is the position of the end boundary in *suffix*.

  `metadata` This specifies a request for information about the state of the current completion. The function should return an alist, as described below. The alist may contain any number of elements.

  If the flag has any other value, the completion function should return `nil`.

The following is a list of metadata entries that a completion function may return in response to a `metadata` flag argument:

`category` The value should be a symbol describing what kind of text the completion function is trying to complete. If the symbol matches one of the keys in `completion-category-overrides`, the usual completion behavior is overridden. See Section 20.6.6 [Completion Variables], page 303.

`annotation-function`
: The value should be a function for *annotating* completions. The function should take one argument, *string*, which is a possible completion. It should return a string, which is displayed after the completion *string* in the '`*Completions*`' buffer.

`display-sort-function`
: The value should be a function for sorting completions. The function should take one argument, a list of completion strings, and return a sorted list of completion strings. It is allowed to alter the input list destructively.

`cycle-sort-function`
: The value should be a function for sorting completions, when `completion-cycle-threshold` is non-`nil` and the user is cycling through completion alternatives. See Section "Completion Options" in *The GNU Emacs Manual*. Its argument list and return value are the same as for `display-sort-function`.

`completion-table-dynamic` *function* [Function]

This function is a convenient way to write a function that can act as a programmed completion function. The argument *function* should be a function that takes one argument, a string, and returns an alist of possible completions of it. You can think of `completion-table-dynamic` as a transducer between that interface and the interface for programmed completion functions.

20.6.8 Completion in Ordinary Buffers

Although completion is usually done in the minibuffer, the completion facility can also be used on the text in ordinary Emacs buffers. In many major modes, in-buffer completion is performed by the *C-M-i* or *M-TAB* command, bound to `completion-at-point`. See Section "Symbol Completion" in *The GNU Emacs Manual*. This command uses the abnormal hook variable `completion-at-point-functions`:

`completion-at-point-functions` [Variable]

The value of this abnormal hook should be a list of functions, which are used to compute a completion table for completing the text at point. It can be used by major modes to provide mode-specific completion tables (see Section 23.2.1 [Major Mode Conventions], page 399).

When the command `completion-at-point` runs, it calls the functions in the list one by one, without any argument. Each function should return `nil` if it is unable to produce a completion table for the text at point. Otherwise it should return a list of the form

```
(start end collection . props)
```

start and *end* delimit the text to complete (which should enclose point). *collection* is a completion table for completing that text, in a form suitable for passing as the second argument to `try-completion` (see Section 20.6.1 [Basic Completion], page 291); completion alternatives will be generated from this completion table in the usual way, via the completion styles defined in `completion-styles` (see Section 20.6.6 [Completion Variables], page 303). *props* is a property list for additional information; any of the properties in `completion-extra-properties` are recognized (see Section 20.6.6 [Completion Variables], page 303), as well as the following additional ones:

`:predicate`
: The value should be a predicate that completion candidates need to satisfy.

`:exclusive`
: If the value is `no`, then if the completion table fails to match the text at point, `completion-at-point` moves on to the next function in `completion-at-point-functions` instead of reporting a completion failure.

A function in `completion-at-point-functions` may also return a function. In that case, that returned function is called, with no argument, and it is entirely responsible for performing the completion. We discourage this usage; it is intended to help convert old code to using `completion-at-point`.

The first function in `completion-at-point-functions` to return a non-`nil` value is used by `completion-at-point`. The remaining functions are not called. The exception to this is when there is an `:exclusive` specification, as described above.

The following function provides a convenient way to perform completion on an arbitrary stretch of text in an Emacs buffer:

`completion-in-region` *start end collection* **&optional** *predicate* [Function]

This function completes the text in the current buffer between the positions *start* and *end*, using *collection*. The argument *collection* has the same meaning as in `try-completion` (see Section 20.6.1 [Basic Completion], page 291).

This function inserts the completion text directly into the current buffer. Unlike `completing-read` (see Section 20.6.2 [Minibuffer Completion], page 294), it does not activate the minibuffer.

For this function to work, point must be somewhere between *start* and *end*.

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 21.5 [Command Loop Info], page 324) 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 either of the mouse or 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-]* 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-]* 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 302, vol. 2), 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 230, vol. 2.

In the following example, the user first types *q*, which is invalid. At the next prompt the user types *y*.

```
(defun ask ()
  (interactive)
  (y-or-n-p "Do you need a lift? "))

;; After evaluation of the preceding definition, M-x ask
;;    causes the following prompt to appear 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* [Function]

Like `y-or-n-p`, except that if the user fails to answer within *seconds* seconds, this function stops waiting and returns *default*. It works by setting up a timer; see Section 39.10 [Timers], page 406, vol. 2. 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 to stop asking questions.

The argument *prompter* specifies how to ask each question. If *prompter* is a string, the question text is computed like this:

```
(format prompter object)
```

where *object* is the next object to ask about (as obtained from *list*).

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 230, vol. 2.

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 21.5 [Command Loop Info], page 324) 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 either of the mouse or 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-event`; see Section 21.5 [Command Loop Info], page 324).

`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).

`previous-complete-history-element` *n* [Command]
: This command replaces the minibuffer contents with the value of the *n*th previous (older) history element that completes the current contents of the minibuffer before the point.

`next-complete-history-element` *n* [Command]
: This command replaces the minibuffer contents with the value of the *n*th next (newer) history element that completes the current contents of the minibuffer before the point.

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 there is none.

`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* is the currently active minibuffer window.

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 156, vol. 2.

`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 294.

`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 21.13 [Recursive Editing], page 355):

`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 316). 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 396.

`minibuffer-exit-hook` [Variable]

This is a normal hook that is run whenever the minibuffer is exited. See Section 23.1 [Hooks], page 396.

`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 457).

`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 that 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* **&rest** *args* [Function]

This function displays *string* temporarily at the end of the minibuffer text, for a few seconds, or until the next input event arrives, whichever comes first. The variable `minibuffer-message-timeout` specifies the number of seconds to wait in the absence of input. It defaults to 2. If *args* is non-`nil`, the actual message is obtained by passing *string* and *args* through `format`. See Section 4.7 [Formatting Strings], page 57.

`minibuffer-inactive-mode` [Command]

This is the major mode used in inactive minibuffers. It uses keymap `minibuffer-inactive-mode-map`. This can be useful if the minibuffer is in a separate frame. See Section 29.8 [Minibuffers and Frames], page 83, vol. 2.

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 input events that translates into a command. It does this by calling the function `read-key-sequence`. Lisp programs can also call this function (see Section 21.8.1 [Key Sequence Input], page 342). They can also read input at a lower level with `read-key` or `read-event` (see Section 21.8.2 [Reading One Event], page 344), or discard pending input with `discard-input` (see Section 21.8.6 [Event Input Misc], page 348).

The key sequence is translated into a command through the currently active keymaps. See Section 22.10 [Key Lookup], page 371, 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 321).

Prior to executing the command, Emacs runs `undo-boundary` to create an undo boundary. See Section 32.10 [Maintaining Undo], page 139, vol. 2.

To execute a command, Emacs first reads its arguments by calling `command-execute` (see Section 21.3 [Interactive Call], page 321). For commands written in Lisp, the `interactive` specification says how to read the arguments. This may use the prefix argument (see Section 21.12 [Prefix Command Arguments], page 353) or may read with prompting in the minibuffer (see Chapter 20 [Minibuffers], page 284). For example, the command `find-file` has an `interactive` specification which says to read a file name using the minibuffer. The function body of `find-file` does not use the minibuffer, so if you call `find-file` as a function from Lisp code, you must supply the file name string as an ordinary Lisp function argument.

If the command is a keyboard macro (i.e. a string or vector), Emacs executes it using `execute-kbd-macro` (see Section 21.16 [Keyboard Macros], page 358).

`pre-command-hook` [Variable]

> This normal hook is run by the editor command loop before it executes each command. At that time, `this-command` contains the command that is about to run, and `last-command` describes the previous command. See Section 21.5 [Command Loop Info], page 324.

`post-command-hook` [Variable]

> This normal hook is run by the editor command loop after it executes each command (including commands terminated prematurely by quitting or by errors). At that time, `this-command` refers to the command that just ran, and `last-command` refers to the command before that.
>
> This hook is also run when Emacs first enters the command loop (at which point `this-command` and `last-command` are both `nil`).

Quitting is suppressed while running `pre-command-hook` and `post-command-hook`. If an error happens while executing one of these hooks, it does not terminate execution of the hook; instead the error is silenced and the function in which the error occurred is removed from the hook.

A request coming into the Emacs server (see Section "Emacs Server" in *The GNU Emacs Manual*) runs these two hooks just as a keyboard command does.

21.2 Defining Commands

The special form `interactive` turns a Lisp function into a command. The `interactive` form must be located at top-level in the function body (usually as the first form in the body), or in the `interactive-form` property of the function symbol. When the `interactive` form is located in the function body, it does nothing when actually executed. Its presence serves as a flag, which tells the Emacs command loop that the function can be called interactively. The argument of the `interactive` form 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 a function is a command, and that it may therefore be called interactively (via *`M-x`* or by entering a key sequence bound to it). The argument *arg-descriptor* declares how to compute the arguments to the command when the command is called interactively.
>
> A command may be called from Lisp programs like any other function, but then the caller supplies the arguments and *arg-descriptor* has no effect.
>
> The `interactive` form must be located at top-level in the function body, or in the function symbol's `interactive-form` property (see Section 8.4.2 [Symbol Plists], page 107). It has its effect because the command loop looks for it before calling the function (see Section 21.3 [Interactive Call], page 321). Once the function is called, all its body forms are executed; at this time, if the `interactive` form occurs within the body, the form simply returns `nil` without even evaluating its argument.
>
> By convention, you should put the `interactive` form in the function body, as the first top-level form. If there is an `interactive` form in both the `interactive-form` symbol property and the function body, the former takes precedence. The `interactive-form` symbol property can be used to add an interactive form to an existing function, or change how its arguments are processed interactively, without redefining the function.

There are three possibilities for the argument *arg-descriptor*:

- It may be omitted or `nil`; then the command is called with no arguments. This leads quickly to an error if the command requires one or more arguments.
- It may be a string; its contents are a sequence of elements separated by newlines, one for each argument[1]. Each element consists of a code character (see Section 21.2.2 [Inter-

[1] Some elements actually supply two arguments.

active Codes], page 318) optionally followed by a prompt (which some code characters use and some ignore). Here is an example:

```
(interactive "P\nbFrobnicate buffer: ")
```

The code letter 'P' sets the command's first argument to the raw command prefix (see Section 21.12 [Prefix Command Arguments], page 353). 'bFrobnicate buffer: ' prompts the user with 'Frobnicate buffer: ' to enter the name of an existing buffer, which becomes the second and final argument.

The prompt string can use '%' to include previous argument values (starting with the first argument) in the prompt. This is done using `format` (see Section 4.7 [Formatting Strings], page 57). For example, here is how you could read the name of an existing buffer followed by a new name to give to that buffer:

```
(interactive "bBuffer to rename: \nsRename buffer %s to: ")
```

If '*' appears at the beginning of the string, then an error is signaled if the buffer is read-only.

If '@' appears at the beginning of the string, and if the key sequence used to invoke the command includes any mouse events, then the window associated with the first of those events is selected before the command is run.

If '^' appears at the beginning of the string, and if the command was invoked through *shift-translation*, set the mark and activate the region temporarily, or extend an already active region, before the command is run. If the command was invoked without shift-translation, and the region is temporarily active, deactivate the region before the command is run. Shift-translation is controlled on the user level by `shift-select-mode`; see Section "Shift Selection" in *The GNU Emacs Manual*.

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 284) or directly from the keyboard (see Section 21.8 [Reading Input], page 342).

  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 321), 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 291). `?` displays a list of possible completions.

Existing Require the name of an existing object. An invalid name is not accepted; the commands to exit the minibuffer do not exit if the current input is not valid.

Default A default value of some sort is used if the user enters no text in the minibuffer. The default depends on the code character.

No I/O This code letter computes an argument without reading any input. Therefore, it does not use a prompt string, and any prompt string you supply is ignored.

Even though the code letter doesn't use a prompt string, you must follow it with a newline if it is not the last code character in the string.

Prompt A prompt immediately follows the code character. The prompt ends either with the end of the string or with a newline.

Special This code character is meaningful only at the beginning of the interactive string, and it does not look for a prompt or a newline. It is a single, isolated character.

Here are the code character descriptions for use with `interactive`:

'`*`' Signal an error if the current buffer is read-only. Special.

'`@`' Select the window mentioned in the first mouse event in the key sequence that invoked this command. Special.

'`^`' If the command was invoked through shift-translation, set the mark and activate the region temporarily, or extend an already active region, before the command is run. If the command was invoked without shift-translation, and the region is temporarily active, deactivate the region before the command is run. Special.

‘`a`’
: A function name (i.e., a symbol satisfying `fboundp`). Existing, Completion, Prompt.

‘`b`’
: The name of an existing buffer. By default, uses the name of the current buffer (see Chapter 27 [Buffers], page 1, vol. 2). 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 99, vol. 2). 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 486). 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 21.7 [Input Events], page 327. 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 *n* events that are lists, the *n*th ‘`e`’ provides the *n*th such event. Events that are not lists, such as function keys and ASCII characters, do not count where ‘`e`’ is concerned.

‘`f`’
: A file name of an existing file (see Section 25.8 [File Names], page 482). The default directory is `default-directory`. Existing, Completion, Default, Prompt.

‘`F`’
: A file name. The file need not exist. Completion, Default, Prompt.

‘`G`’
: A file name. The file need not exist. If the user enters just a directory name, then the value is just that directory name, with no file name within the directory added. Completion, Default, Prompt.

‘`i`’
: An irrelevant argument. This code always supplies `nil` as the argument’s value. No I/O.

‘`k`’
: A key sequence (see Section 22.1 [Key Sequences], page 360). This keeps reading events until a command (or undefined command) is found in the current key maps. The key sequence argument is represented as a string or vector. The cursor does not move into the echo area. Prompt.

  If ‘`k`’ reads a key sequence that ends with a down-event, it also reads and discards the following up-event. You can get access to that up-event with the ‘`U`’ code character.

  This kind of input is used by commands such as `describe-key` and `global-set-key`.

‘`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 *The GNU Emacs Manual*). Prompt.
‘`n`’	A number, read with the minibuffer. If the input is not a number, the user has to try again. ‘`n`’ never uses the prefix argument. Prompt.
‘`N`’	The numeric prefix argument; but if there is no prefix argument, read a number as with *n*. The value is always a number. See Section 21.12 [Prefix Command Arguments], page 353. 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 285). Terminate the input with either *C-j* or `RET`. (*C-q* may be used to include either of these characters in the input.) Prompt.
‘`S`’	An interned symbol whose name is read in the minibuffer. Any whitespace character terminates the input. (Use *C-q* 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 `nil`. Can be used after a ‘`k`’ or ‘`K`’ argument to get the up-event that was discarded (if any) after ‘`k`’ or ‘`K`’ read a down-event. If no up-event has been discarded, ‘`U`’ provides `nil` as the argument. No I/O.
‘`v`’	A variable declared to be a user option (i.e., satisfying the predicate `user-variable-p`). This reads the variable using `read-variable`. See [Definition of read-variable], page 300. Existing, Completion, Prompt.
‘`x`’	A Lisp object, specified with its read syntax, terminated with a *C-j* or `RET`. The object is not evaluated. See Section 20.3 [Object from Minibuffer], page 288. Prompt.
‘`X`’	A Lisp form’s value. ‘`X`’ reads as ‘`x`’ does, then evaluates the form so that its value becomes the argument for the command. Prompt.
‘`z`’	A coding system name (a symbol). If the user enters null input, the argument value is `nil`. See Section 33.9 [Coding Systems], page 193, vol. 2. Completion, Existing, Prompt.
‘`Z`’	A coding system name (a symbol)—but only if this command has a prefix argument. With no prefix argument, ‘`Z`’ provides `nil` as the argument value. Completion, Existing, Prompt.

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.
                            ; under shift-select-mode,
                            ;     will activate or extend region.
    (forward-word (* 2 n)))
     ⇒ foo2

(defun foo3 (n)             ; foo3 takes one argument,
    (interactive "nCount:") ;     which is read with the Minibuffer.
    (forward-word (* 2 n)))
     ⇒ foo3

(defun three-b (b1 b2 b3)
  "Select three existing buffers.
Put them into three windows, selecting the last one."
    (interactive "bBuffer1:\nbBuffer2:\nbBuffer3:")
    (delete-other-windows)
    (split-window (selected-window) 8)
    (switch-to-buffer b1)
    (other-window 1)
    (split-window (selected-window) 8)
    (switch-to-buffer b2)
    (other-window 1)
    (switch-to-buffer b3))
     ⇒ three-b
(three-b "*scratch*" "declarations.texi" "*mail*")
     ⇒ nil
```

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.

Note that the term “command”, in this context, refers to an interactively callable function (or function-like object), or a keyboard macro. It does not refer to the key sequence used to invoke a command (see Chapter 22 [Keymaps], page 360).

`commandp` *object* **&optional** *for-call-interactively* [Function]

This function returns `t` if *object* is a command. Otherwise, it returns `nil`.

Commands include strings and vectors (which are treated as keyboard macros), lambda expressions that contain a top-level `interactive` form (see Section 21.2.1 [Using Interactive], page 316), byte-code function objects made from such lambda expressions, autoload objects that are declared as interactive (non-`nil` fourth argument to `autoload`), and some primitive functions. Also, a symbol is considered a command if it has a non-`nil` `interactive-form` property, or if its function definition satisfies `commandp`.

If *for-call-interactively* is non-`nil`, then `commandp` returns `t` only for objects that `call-interactively` could call—thus, not for keyboard macros.

See `documentation` in Section 24.2 [Accessing Documentation], page 452, for a realistic example of using `commandp`.

`call-interactively` *command* **&optional** *record-flag keys* [Function]

This function calls the interactively callable function *command*, providing arguments according to its interactive calling specifications. It returns whatever *command* returns.

If, for instance, you have a function with the following signature:

```
(defun foo (begin end)
  (interactive "r")
  ...)
```

then saying

```
(call-interactively 'foo)
```

will call `foo` with the region (`point` and `mark`) as the arguments.

An error is signaled if *command* is not a function or if it cannot be called interactively (i.e., is not a command). Note that keyboard macros (strings and vectors) are not accepted, even though they are considered commands, because they are not functions. If *command* is a symbol, then `call-interactively` uses its function definition.

If *record-flag* is non-`nil`, then this command and its arguments are unconditionally added to the list `command-history`. Otherwise, the command is added only if it uses the minibuffer to read an argument. See Section 21.15 [Command History], page 357.

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 326.

`command-execute` *command* **&optional** *record-flag keys special* [Function]

This function executes *command*. The argument *command* must satisfy the `commandp` predicate; i.e., it must be an interactively callable function or a keyboard macro.

A string or vector as *command* is executed with `execute-kbd-macro`. A function is passed to `call-interactively` (see above), along with the *record-flag* and *keys* arguments.

If *command* is a symbol, its function definition is used in its place. A symbol with an `autoload` definition counts as a command if it was declared to stand for an interactively callable function. Such a definition is handled by loading the specified library and then rechecking the definition of the symbol.

The argument *special*, if given, means to ignore the prefix argument and not clear it. This is used for executing special events (see Section 21.9 [Special Events], page 350).

`execute-extended-command` *prefix-argument* [Command]

This function reads a command name from the minibuffer using `completing-read` (see Section 20.6 [Completion], page 291). 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
```

21.4 Distinguish Interactive Calls

Sometimes a command should display additional visual feedback (such as an informative message in the echo area) for interactive calls only. There are three ways to do this. The recommended way to test whether the function was called using `call-interactively` is to give it an optional argument `print-message` and use the `interactive` spec to make it non-`nil` in interactive calls. Here's an example:

```
(defun foo (&optional print-message)
  (interactive "p")
  (when print-message
    (message "foo")))
```

We use `"p"` because the numeric prefix argument is never `nil`. Defined in this way, the function does display the message when called from a keyboard macro.

The above method with the additional argument is usually best, because it allows callers to say "treat this call as interactive". But you can also do the job by testing `called-interactively-p`.

`called-interactively-p` *kind* [Function]

This function returns `t` when the calling function was called using `call-interactively`.

The argument *kind* should be either the symbol `interactive` or the symbol `any`. If it is `interactive`, then `called-interactively-p` returns `t` only if the call was made directly by the user—e.g., if the user typed a key sequence bound to the calling function, but *not* if the user ran a keyboard macro that called the function (see Section 21.16 [Keyboard Macros], page 358). If *kind* is `any`, `called-interactively-p` returns `t` for any kind of interactive call, including keyboard macros.

If in doubt, use `any`; the only known proper use of `interactive` is if you need to decide whether to display a helpful message while a function is running.

A function is never considered to be called interactively if it was called via Lisp evaluation (or with `apply` or `funcall`).

Here is an example of using `called-interactively-p`:

```
(defun foo ()
  (interactive)
  (when (called-interactively-p 'any)
    (message "Interactive!")
    'foo-called-interactively))

;; Type M-x foo.
     ⊣ Interactive!

(foo)
     ⇒ nil
```

Here is another example that contrasts direct and indirect calls to `called-interactively-p`.

```
(defun bar ()
  (interactive)
  (message "%s" (list (foo) (called-interactively-p 'any))))

;; Type M-x bar.
     ⊣ (nil t)
```

21.5 Information from the Command Loop

The editor command loop sets several Lisp variables to keep status records for itself and for commands that are run. With the exception of `this-command` and `last-command` it's generally a bad idea to change any of these variables in a Lisp program.

`last-command` [Variable]

This variable records the name of the previous command executed by the command loop (the one before the current command). Normally the value is a symbol with a function definition, but this is not guaranteed.

The value is copied from `this-command` when a command returns to the command loop, except when the command has specified a prefix argument for the following command.

This variable is always local to the current terminal and cannot be buffer-local. See Section 29.2 [Multiple Terminals], page 67, vol. 2.

`real-last-command` [Variable]

This variable is set up by Emacs just like `last-command`, but never altered by Lisp programs.

`last-repeatable-command` [Variable]

This variable stores the most recently executed command that was not part of an input event. This is the command `repeat` will try to repeat, See Section "Repeating" in *The GNU Emacs Manual*.

`this-command` [Variable]

This variable records the name of the command now being executed by the editor command loop. Like `last-command`, it is normally a symbol with a function definition.

The command loop sets this variable just before running a command, and copies its value into `last-command` when the command finishes (unless the command specified a prefix argument for the following command).

Some commands set this variable during their execution, as a flag for whatever command runs next. In particular, the functions for killing text set `this-command` to `kill-region` so that any kill commands immediately following will know to append the killed text to the previous kill.

If you do not want a particular command to be recognized as the previous command in the case where it got an error, you must code that command to prevent this. One way is to set `this-command` to `t` at the beginning of the command, and set `this-command` back to its proper value at the end, like this:

```
(defun foo (args...)
  (interactive ...)
  (let ((old-this-command this-command))
    (setq this-command t)
    ...do the work...
    (setq this-command old-this-command)))
```

We do not bind `this-command` with `let` because that would restore the old value in case of error—a feature of `let` which in this case does precisely what we want to avoid.

`this-original-command` [Variable]

This has the same value as `this-command` except when command remapping occurs (see Section 22.13 [Remapping Commands], page 378). 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 21.8.1 [Key Sequence Input], page 342. The value is a string if all events in the sequence were characters that fit in a string. See Section 21.7 [Input Events], page 327.

```
(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 21.7.15 [Strings of Events], page 341).

`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 410, vol. 2) 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 307).

`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` is obsolete.

`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 83, vol. 2.

If the last event came from a keyboard macro, the value is `macro`.

21.6 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 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.

21.7 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`.

21.7.1 Keyboard Events

There are two kinds of input you can get from the keyboard: ordinary keys, and function keys. Ordinary keys correspond to 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 21.7.12 [Classifying Events], page 336.

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 *C-A* from *C-a*, so Emacs uses the 2^{25} bit in *C-A* and not in *C-a*.

hyper The 2^{24} bit in the character 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. (The key labeled `Alt` on most keyboards is actually treated as the meta key, not this.)

It is best to avoid mentioning specific bit numbers in your program. To test the modifier bits of a character, use the function `event-modifiers` (see Section 21.7.12 [Classifying Events], page 336). 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 375). The function `event-convert-list` converts such a list into an event type (see Section 21.7.12 [Classifying Events], page 336).

21.7.2 Function Keys

Most keyboards also have *function keys*—keys that have names or symbols that are not characters. Function keys are represented in Emacs Lisp as symbols; the symbol’s name is the function key’s label, in lower case. For example, pressing a key labeled `F1` generates an input event represented by the symbol `f1`.

The event type of a function key event is the event symbol itself. See Section 21.7.12 [Classifying Events], page 336.

Here are a few special cases in the symbol-naming convention for function keys:

`backspace`, `tab`, `newline`, `return`, `delete`
: These keys correspond to common ASCII control characters that have special keys on most keyboards.

  In ASCII, *`C-i`* and `TAB` are the same character. If the terminal can distinguish between them, Emacs conveys the distinction to Lisp programs by representing the former as the integer 9, and the latter as the symbol `tab`.

  Most of the time, it’s not useful to distinguish the two. So normally `local-function-key-map` (see Section 22.14 [Translation Keymaps], page 378) 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.

21.7.3 Mouse Events

Emacs supports four kinds of mouse events: click events, drag events, button-down events, and motion events. All mouse events are represented as lists. The CAR of the list is the event type; this says which mouse button was involved, and which modifier keys were used with it. The event type can also distinguish double or triple button presses (see Section 21.7.7 [Repeat Events], page 332). 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 318.

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.

21.7.4 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.

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))
```

The meanings of these list elements are documented below. See Section 21.7.13 [Accessing Mouse], page 338, for functions that let you easily access these elements.

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`.

In one special case, *pos-or-area* is a list containing a symbol (one of the symbols listed above) instead of just the symbol. This happens after the imaginary prefix keys for the event are registered by Emacs. See Section 21.8.1 [Key Sequence Input], page 342.

x, *y* These are the relative pixel coordinates of the click. For clicks in the text area of a window, the coordinate origin `(0 . 0)` is taken to be the top left corner of the text area. See Section 28.3 [Window Sizes], page 22, vol. 2. For clicks in a mode line or header line, the coordinate origin is the top left corner of the window itself. For fringes, margins, and the vertical border, *x* does not have meaningful data. For fringes and margins, *y* is relative to the bottom edge of the header line. In all cases, the *x* and *y* coordinates increase rightward and downward respectively.

timestamp

This is the time at which the event occurred, in milliseconds.

object This is either `nil` if there is no string-type text property at the click position, or a cons cell of the form (*string* . *string-pos*) if there is one:

string The string which was clicked on, including any properties.

string-pos The position in the string where the click occurred.

text-pos For clicks on a marginal area or on a fringe, this is the buffer position of the first visible character in the corresponding line in the window. For other events, it is the current buffer position in the window.

col, *row* These are the actual column and row coordinate numbers of the glyph under the *x*, *y* position. If *x* lies beyond the last column of actual text on its line, *col* is reported by adding fictional extra columns that have the default character width. Row 0 is taken to be the header line if the window has one, or the topmost row of the text area otherwise. Column 0 is taken to be the leftmost column of the text area for clicks on a window text area, or the leftmost mode line or header line column for clicks there. For clicks on fringes or vertical borders, these have no meaningful data. For clicks on margins, *col* is measured from the left edge of the margin area and *row* is measured from the top of the margin area.

image This is the image object on which the click occurred. It is either `nil` if there is no image at the position clicked on, or it is an image object as returned by `find-image` if click was in an image.

dx, *dy* These are the pixel coordinates of the click, relative to the top left corner of *object*, which is `(0 . 0)`. If *object* is `nil`, the coordinates are relative to the top left corner of the character glyph clicked on.

width, *height*
These are the pixel width and height of *object* or, if this is `nil`, those of the character glyph clicked on.

For mouse clicks on a scroll-bar, *position* has this form:

```
(window area (portion . whole) timestamp part)
```

window This is the window whose scroll-bar was clicked on.

area This is the scroll bar where the click occurred. It is one of the symbols `vertical-scroll-bar` or `horizontal-scroll-bar`.

portion This is the distance of the click from the top or left end of the scroll bar.

whole This is the length of the entire scroll bar.

timestamp
This is the time at which the event occurred, in milliseconds.

part This is the part of the scroll-bar which was clicked on. It is one of the symbols `above-handle`, `handle`, `below-handle`, `up`, `down`, `top`, `bottom`, and `end-scroll`.

click-count
This is the number of rapid repeated presses so far of the same mouse button. See Section 21.7.7 [Repeat Events], page 332.

21.7.5 Drag Events

With Emacs, you can have a drag event without even changing your clothes. A *drag event* happens every time the user presses a mouse button and then moves the mouse to a different character position before releasing the button. Like all mouse events, drag events are represented in Lisp as lists. The lists record both the starting mouse position and the final position, like this:

```
(event-type
 (window1 START-POSITION)
 (window2 END-POSITION))
```

For a drag event, the name of the symbol *event-type* contains the prefix '`drag-`'. For example, dragging the mouse with button 2 held down generates a `drag-mouse-2` event. The second and third elements of the event give the starting and ending position of the drag. They have the same form as *position* in a click event (see Section 21.7.4 [Click Events], page 329) that is not on the scroll bar part of the window. 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.

21.7.6 Button-Down Events

Click and drag events happen when the user releases a mouse button. They cannot happen earlier, because there is no way to distinguish a click from a drag until the button is released.

If you want to take action as soon as a button is pressed, you need to handle *button-down* events.[2] These occur as soon as a button is pressed. They are represented by lists that look exactly like click events (see Section 21.7.4 [Click Events], page 329), 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 21.7.8 [Motion Events], page 334.

21.7.7 Repeat Events

If you press the same mouse button more than once in quick succession without moving the mouse, Emacs generates special *repeat* mouse events for the second and subsequent presses.

The most common repeat events are *double-click* events. Emacs generates a double-click event when you click a button twice; the event happens when you release the button (as is normal for all click events).

[2] Button-down is the conservative antithesis of drag.

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.

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.

21.7.8 Motion Events

Emacs sometimes generates *mouse motion* events to describe motion of the mouse without any button activity. Mouse motion events are represented by lists that look like this:

```
(mouse-movement POSITION)
```

The second element of the list describes the current position of the mouse, just as in a click event (see Section 21.7.4 [Click Events], page 329).

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 87, vol. 2.

21.7.9 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 83, vol. 2.

Focus events are represented in Lisp as lists that look like this:

```
(switch-frame new-frame)
```

where *new-frame* is the frame switched to.

Some X window managers are set up so that just moving the mouse into a window is enough to set the focus there. Usually, there is no need for a Lisp program to know about the focus change until some other kind of input arrives. Emacs generates a focus event only when the user actually types a keyboard key or presses a mouse button in the new frame; just moving the mouse between frames does not generate a focus event.

A focus event in the middle of a key sequence would garble the sequence. So Emacs never generates a focus event in the middle of a key sequence. If the user changes focus in the middle of a key sequence—that is, after a prefix key—then Emacs reorders the events so that the focus event comes either before or after the multi-event key sequence, and not within it.

21.7.10 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 (see Section 21.7.4 [Click Events], page 329).

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 (see Section 21.7.4 [Click Events], page 329), 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 163, vol. 2.

`sigusr1`
`sigusr2`
: These events are generated when the Emacs process receives the signals `SIGUSR1` and `SIGUSR2`. They contain no additional data because signals do not carry additional information. They can be useful for debugging (see Section 18.1.1 [Error Debugging], page 243).

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 367). 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.

21.7.11 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)
```

21.7.12 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
```

21.7.13 Accessing Mouse 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 (see Section 21.7.4 [Click Events], page 329):

```
(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 the window-relative coordinates in the text area of a window into frame-relative coordinates:

```
(defun frame-relative-coordinates (position)
  "Return frame-relative coordinates from POSITION.
POSITION is assumed to lie in a window text area."
  (let* ((x-y (posn-x-y position))
         (window (posn-window position))
         (edges (window-inside-pixel-edges window)))
    (cons (+ (car x-y) (car edges))
          (+ (cdr x-y) (cadr edges)))))
```

posn-col-row *position* [Function]

This function returns a cons cell `(col . row)`, containing the estimated column and row corresponding to buffer position *position*. The return value is given in units of the frame's default character width and height, as computed from the *x* and *y* values corresponding to *position*. (So, if the actual characters have non-default sizes, the actual row and column may differ from these computed values.)

Note that *row* is counted from the top of the text area. If the window possesses a header line (see Section 23.4.7 [Header Lines], page 426), it is *not* counted as the first line.

posn-actual-col-row *position* [Function]

Return the actual row and column in *position*, as a cons cell `(col . row)`. The values are the actual row and column numbers in the window. See Section 21.7.4 [Click Events], page 329, for details. It returns `nil` if *position* does not include actual positions 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.

21.7.14 Accessing Scroll Bar Events

These functions are useful for decoding scroll bar events.

`scroll-bar-event-ratio` *event* [Function]
: This function returns the fractional vertical position of a scroll bar event within the scroll bar. The value is a cons cell `(`***`portion`*** `.` ***`whole`***`)` containing two integers whose ratio is the fractional position.

`scroll-bar-scale` *ratio total* [Function]
: This function multiplies (in effect) *ratio* by *total*, rounding the result to an integer. The argument *ratio* is not a number, but rather a pair `(`***`num`*** `.` ***`denom`***`)`—typically a value returned by `scroll-bar-event-ratio`.

  This function is handy for scaling a position on a scroll bar into a buffer position. Here's how to do that:

```
          (+ (point-min)
             (scroll-bar-scale
                (posn-x-y (event-start event))
                (- (point-max) (point-min))))
```

Recall that scroll bar events have two integers forming a ratio, in place of a pair of x and y coordinates.

21.7.15 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 21.8.6 [Event Input Misc], page 348) 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.

21.8 Reading Input

The editor command loop reads key sequences using the function `read-key-sequence`, which uses `read-event`. These and other functions for event input are also available for use in Lisp programs. See also `momentary-string-display` in Section 38.8 [Temporary Displays], page 313, vol. 2, and `sit-for` in Section 21.10 [Waiting], page 350. See Section 39.12 [Terminal Input], page 409, vol. 2, for functions and variables for controlling terminal input modes and debugging terminal input.

For higher-level input facilities, see Chapter 20 [Minibuffers], page 284.

21.8.1 Key Sequence Input

The command loop reads input a key sequence at a time, by calling `read-key-sequence`. Lisp programs can also call this function; for example, `describe-key` uses it to read the key to describe.

`read-key-sequence` *prompt* **&optional** *continue-echo dont-downcase-last switch-frame-ok command-loop* [Function]

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 21.7.15 [Strings of Events], page 341). 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 378.

The argument *prompt* is either a string to be displayed in the echo area as a prompt, or `nil`, meaning not to display a prompt. The argument *continue-echo*, if non-`nil`, means to echo this key as a continuation of the previous key.

Normally any upper case event is converted to lower case if the original event is undefined and the lower case equivalent is defined. The argument *dont-downcase-last*, if non-`nil`, means do not convert the last event to lower case. This is appropriate for reading a key sequence to be defined.

The argument *switch-frame-ok*, if non-`nil`, means that this function should process a `switch-frame` event if the user switches frames before typing anything. If the user switches frames in the middle of a key sequence, or at the start of the sequence but *switch-frame-ok* is `nil`, then the event will be put off until after the current key sequence.

The argument *command-loop*, if non-`nil`, means that this key sequence is being read by something that will read commands one after another. It should be `nil` if the caller will read just one key sequence.

In the following example, Emacs displays the prompt '?' in the echo area, and then the user types *C-x C-f*.

```
(read-key-sequence "?")

---------- Echo Area ----------
?C-x C-f
---------- Echo Area ----------

     ⇒ "^X^F"
```

The function `read-key-sequence` suppresses quitting: *C-g* typed while reading with this function works like any other character, and does not set `quit-flag`. See Section 21.11 [Quitting], page 351.

`read-key-sequence-vector` *prompt* **&optional** *continue-echo dont-downcase-last switch-frame-ok command-loop* [Function]

This is like `read-key-sequence` except that it always returns the key sequence as a vector, never as a string. See Section 21.7.15 [Strings of Events], page 341.

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.

When reading input results in such a *shift-translation*, Emacs sets the variable `this-command-keys-shift-translated` to a non-`nil` value. Lisp programs can examine this variable if they need to modify their behavior when invoked by shift-translated keys. For example, the function `handle-shift-selection` examines the value of this variable to determine how to activate or deactivate the region (see Section 31.7 [The Mark], page 117, vol. 2).

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.

21.8.2 Reading One Event

The lowest level functions for command input are `read-event`, `read-char`, and `read-char-exclusive`.

`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 302, vol. 2.

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 408, vol. 2)—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 457. Certain other events, called *special events*, are also processed directly within `read-event` (see Section 21.9 [Special Events], page 350).

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`.

None of the above functions suppress quitting.

`num-nonmacro-input-events` [Variable]

This variable holds the total number of input events received so far from the terminal—not counting those generated by keyboard macros.

We emphasize that, unlike `read-key-sequence`, the functions `read-event`, `read-char`, and `read-char-exclusive` do not perform the translations described in Section 22.14 [Translation Keymaps], page 378. If you wish to read a single key taking these translations into account, use the function `read-key`:

`read-key` **&optional** *prompt* [Function]

This function reads a single key. It is "intermediate" between `read-key-sequence` and `read-event`. Unlike the former, it reads a single key, not a key sequence. Unlike the latter, it does not return a raw event, but decodes and translates the user input according to `input-decode-map`, `local-function-key-map`, and `key-translation-map` (see Section 22.14 [Translation Keymaps], page 378).

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.

`read-char-choice` *prompt chars* **&optional** *inhibit-quit* [Function]

This function uses `read-key` to read and return a single character. It ignores any input that is not a member of *chars*, a list of accepted characters. Optionally, it will also ignore keyboard-quit events while it is waiting for valid input. If you bind `help-form` (see Section 24.5 [Help Functions], page 457) to a non-`nil` value while calling `read-char-choice`, then pressing `help-char` causes it to evaluate `help-form` and display the result. It then continues to wait for a valid input character, or keyboard-quit.

21.8.3 Modifying and Translating Input Events

Emacs modifies every event it reads according to `extra-keyboard-modifiers`, then translates it through `keyboard-translate-table` (if applicable), before returning it from `read-event`.

`extra-keyboard-modifiers` [Variable]

This variable lets Lisp programs "press" the modifier keys on the keyboard. The value is a character. Only the modifiers of the character matter. Each time the user types a keyboard key, it is altered as if those modifier keys were held down. For instance, if you bind `extra-keyboard-modifiers` to `?\C-\M-a`, then all keyboard input characters typed during the scope of the binding will have the control and meta modifiers applied to them. The character `?\C-@`, equivalent to the integer 0, does not count as a control character for this purpose, but as a character with no modifiers. Thus, setting `extra-keyboard-modifiers` to zero cancels any modification.

When using a window system, the program can "press" any of the modifier keys in this way. Otherwise, only the `CTL` and `META` keys can be virtually pressed.

Note that this variable applies only to events that really come from the keyboard, and has no effect on mouse events or any other events.

`keyboard-translate-table` [Variable]

This terminal-local variable is the translate table for keyboard characters. It lets you reshuffle the keys on the keyboard without changing any command bindings. Its value is normally a char-table, or else `nil`. (It can also be a string or vector, but this is considered obsolete.)

If `keyboard-translate-table` is a char-table (see Section 6.6 [Char-Tables], page 92), 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 33.10 [Input Methods], page 206, vol. 2). Use `translation-table-for-input` (see Section 33.8 [Translation of Characters], page 191, vol. 2), 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 378, for mechanisms that translate event sequences at the level of `read-key-sequence`.

21.8.4 Invoking the Input Method

The event-reading functions invoke the current input method, if any (see Section 33.10 [Input Methods], page 206, vol. 2). 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 21.8.6 [Event Input Misc], page 348). 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.

21.8.5 Quoted Character Input

You can use the function `read-quoted-char` to ask the user to specify a character, and allow the user to specify a control or meta character conveniently, either literally or as an octal character code. The command `quoted-insert` uses this function.

`read-quoted-char` **&optional** *prompt* [Function]

This function is like `read-char`, except that if the first character read is an octal digit (0-7), it reads any number of octal digits (but stopping if a non-octal digit is found), and returns the character represented by that numeric character code. If the character that terminates the sequence of octal digits is `RET`, it is discarded. Any other terminating character is used as input after this function returns.

Quitting is suppressed when the first character is read, so that the user can enter a *C-g*. See Section 21.11 [Quitting], page 351.

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
```

21.8.6 Miscellaneous Event Input Features

This section describes how to "peek ahead" at events without using them up, how to check for pending input, and how to discard pending input. See also the function `read-passwd` (see Section 20.9 [Reading a Password], page 310).

`unread-command-events` [Variable]

This variable holds a list of events waiting to be read as command input. The events are used in the order they appear in the list, and removed one by one as they are used.

The variable is needed because in some cases a function reads an event and then decides not to use it. Storing the event in this variable causes it to be processed normally, by the command loop or by the functions to read command input.

For example, the function that implements numeric prefix arguments reads any number of digits. When it finds a non-digit event, it must unread the event so that it can be read normally by the command loop. Likewise, incremental search uses this feature to unread events with no special meaning in a search, because these events should exit the search and then execute normally.

The reliable and easy way to extract events from a key sequence so as to put them in `unread-command-events` is to use `listify-key-sequence` (see below).

Normally you add events to the front of this list, so that the events most recently unread will be reread first.

Events read from this list are not normally added to the current command's key sequence (as returned by e.g. `this-command-keys`), as the events will already have been added once as they were read for the first time. An element of the form `(t . event)` forces *event* to be added to the current command's key sequence.

`listify-key-sequence` *key* [Function]

This function converts the string or vector *key* to a list of individual events, which you can put in `unread-command-events`.

`input-pending-p` [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` is obsolete.

`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
```

21.9 Special Events

Certain *special events* are handled at a very low level—as soon as they are read. The `read-event` function processes these events itself, and never returns them. Instead, it keeps waiting for the first event that is not special and returns that one.

Special events do not echo, they are never grouped into key sequences, and they never appear in the value of `last-command-event` or `(this-command-keys)`. They do not discard a numeric argument, they cannot be unread with `unread-command-events`, they may not appear in a keyboard macro, and they are not recorded in a keyboard macro while you are defining one.

Special events do, however, appear in `last-input-event` immediately after they are read, and this is the way for the event's definition to find the actual event.

The events types `iconify-frame`, `make-frame-visible`, `delete-frame`, `drag-n-drop`, 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 367).

21.10 Waiting for Elapsed Time or Input

The wait functions are designed to wait for a certain amount of time to pass or until there is input. For example, you may wish to pause in the middle of a computation to allow the user time to view the display. `sit-for` pauses and updates the screen, and returns immediately if input comes in, while `sleep-for` pauses without updating the screen.

`sit-for` *seconds* **&optional** *nodisp* [Function]

This function performs redisplay (provided there is no pending input from the user), then waits *seconds* seconds, or until input is available. The usual purpose of `sit-for` is to give the user time to read text that you display. The value is `t` if `sit-for` waited the full time with no input arriving (see Section 21.8.6 [Event Input Misc], page 348). 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 299, vol. 2.

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 413, vol. 2), `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 399, vol. 2, for functions to get the current time.

21.11 Quitting

Typing *`C-g`* while a Lisp function is running causes Emacs to *quit* whatever it is doing. This means that control returns to the innermost active command loop.

Typing *`C-g`* while the command loop is waiting for keyboard input does not cause a quit; it acts as an ordinary input character. In the simplest case, you cannot tell the difference, because *`C-g`* normally runs the command `keyboard-quit`, whose effect is to quit. However, when *`C-g`* follows a prefix key, they combine to form an undefined key. The effect is to cancel the prefix key as well as any prefix argument.

In the minibuffer, *`C-g`* has a different definition: it aborts out of the minibuffer. This means, in effect, that it exits the minibuffer and then quits. (Simply quitting would return to the command loop *within* the minibuffer.) The reason why *`C-g`* does not quit directly when the command reader is reading input is so that its meaning can be redefined in the minibuffer in this way. *`C-g`* following a prefix key is not redefined in the minibuffer, and it has its normal effect of canceling the prefix key and prefix argument. This too would not be possible if *`C-g`* always quit directly.

When *`C-g`* does directly quit, it does so by setting the variable `quit-flag` to `t`. Emacs checks this variable at appropriate times and quits if it is not `nil`. Setting `quit-flag` non-`nil` in any way thus causes a quit.

At the level of C code, quitting cannot happen just anywhere; only at the special places that check `quit-flag`. The reason for this is that quitting at other places might leave an inconsistency in Emacs's internal state. Because quitting is delayed until a safe place, quitting cannot make Emacs crash.

Certain functions such as `read-key-sequence` or `read-quoted-char` prevent quitting entirely even though they wait for input. Instead of quitting, *C-g* serves as the requested input. In the case of `read-key-sequence`, this serves to bring about the special behavior of *C-g* in the command loop. In the case of `read-quoted-char`, this is so that *C-q* can be used to quote a *C-g*.

You can prevent quitting for a portion of a Lisp function by binding the variable `inhibit-quit` to a non-`nil` value. Then, although *C-g* still sets `quit-flag` to `t` as usual, the usual result of this—a quit—is prevented. Eventually, `inhibit-quit` will become `nil` again, such as when its binding is unwound at the end of a `let` form. At that time, if `quit-flag` is still non-`nil`, the requested quit happens immediately. This behavior is ideal when you wish to make sure that quitting does not happen within a "critical section" of the program.

In some functions (such as `read-quoted-char`), *C-g* is handled in a special way that does not involve quitting. This is done by reading the input with `inhibit-quit` bound to `t`, and setting `quit-flag` to `nil` before `inhibit-quit` becomes `nil` again. This excerpt from the definition of `read-quoted-char` shows how this is done; it also shows that normal quitting is permitted after the first character of input.

```
(defun read-quoted-char (&optional prompt)
  "...documentation..."
  (let ((message-log-max nil) done (first t) (code 0) char)
    (while (not done)
      (let ((inhibit-quit first)
            ...)
        (and prompt (message "%s-" prompt))
        (setq char (read-event))
        (if inhibit-quit (setq quit-flag nil)))
      ...set the variable code...)
    code))
```

`quit-flag` [Variable]

> If this variable is non-`nil`, then Emacs quits immediately, unless `inhibit-quit` is non-`nil`. Typing *C-g* ordinarily sets `quit-flag` non-`nil`, regardless of `inhibit-quit`.

`inhibit-quit` [Variable]

> This variable determines whether Emacs should quit when `quit-flag` is set to a value other than `nil`. If `inhibit-quit` is non-`nil`, then `quit-flag` has no special effect.

`with-local-quit` *body...* [Macro]

> This macro executes *body* forms in sequence, but allows quitting, at least locally, within *body* even if `inhibit-quit` was non-`nil` outside this construct. It returns the value of the last form in *body*, unless exited by quitting, in which case it returns `nil`.
>
> If `inhibit-quit` is `nil` on entry to `with-local-quit`, it only executes the *body*, and setting `quit-flag` causes a normal quit. However, if `inhibit-quit` is non-`nil` so that ordinary quitting is delayed, a non-`nil` `quit-flag` triggers a special kind of local quit. This ends the execution of *body* and exits the `with-local-quit` body

with `quit-flag` still non-`nil`, so that another (ordinary) quit will happen as soon as that is allowed. If `quit-flag` is already non-`nil` at the beginning of *body*, the local quit happens immediately and the body doesn't execute at all.

This macro is mainly useful in functions that can be called from timers, process filters, process sentinels, `pre-command-hook`, `post-command-hook`, and other places where `inhibit-quit` is normally bound to `t`.

`keyboard-quit` [Command]

This function signals the `quit` condition with `(signal 'quit nil)`. This is the same thing that quitting does. (See `signal` in Section 10.5.3 [Errors], page 128.)

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 409, vol. 2.

21.12 Prefix Command Arguments

Most Emacs commands can use a *prefix argument*, a number specified before the command itself. (Don't confuse prefix arguments with prefix keys.) The prefix argument is at all times represented by a value, which may be `nil`, meaning there is currently no prefix argument. Each command may use the prefix argument or ignore it.

There are two representations of the prefix argument: *raw* and *numeric*. The editor command loop uses the raw representation internally, and so do the Lisp variables that store the information, but commands can request either representation.

Here are the possible values of a raw prefix argument:

- `nil`, meaning there is no prefix argument. Its numeric value is 1, but numerous commands make a distinction between `nil` and the integer 1.
- An integer, which stands for itself.
- A list of one element, which is an integer. This form of prefix argument results from one or a succession of *C-u*s with no digits. The numeric value is the integer in the list, but some commands make a distinction between such a list and an integer alone.
- The symbol `-`. This indicates that *M--* or *C-u -* was typed, without following digits. The equivalent numeric value is −1, but some commands make a distinction between the integer −1 and the symbol `-`.

We illustrate these possibilities by calling the following function with various prefixes:

```
(defun display-prefix (arg)
  "Display the value of the raw prefix arg."
  (interactive "P")
  (message "%s" arg))
```

Here are the results of calling `display-prefix` with various raw prefix arguments:

```
        M-x display-prefix  ⊣ nil

C-u     M-x display-prefix  ⊣ (4)

C-u C-u M-x display-prefix  ⊣ (16)
```

```
C-u 3   M-x display-prefix  ⊣ 3

M-3     M-x display-prefix  ⊣ 3      ; (Same as C-u 3.)

C-u -   M-x display-prefix  ⊣ -

M--     M-x display-prefix  ⊣ -      ; (Same as C-u -.)

C-u - 7 M-x display-prefix  ⊣ -7

M-- 7   M-x display-prefix  ⊣ -7     ; (Same as C-u -7.)
```

Emacs uses two variables to store the prefix argument: `prefix-arg` and `current-prefix-arg`. Commands such as `universal-argument` that set up prefix arguments for other commands store them in `prefix-arg`. In contrast, `current-prefix-arg` conveys the prefix argument to the current command, so setting it has no effect on the prefix arguments for future commands.

Normally, commands specify which representation to use for the prefix argument, either numeric or raw, in the `interactive` specification. (See Section 21.2.1 [Using Interactive], page 316.) 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.

21.13 Recursive Editing

The Emacs command loop is entered automatically when Emacs starts up. This top-level invocation of the command loop never exits; it keeps running as long as Emacs does. Lisp programs can also invoke the command loop. Since this makes more than one activation of the command loop, we call it *recursive editing*. A recursive editing level has the effect of suspending whatever command invoked it and permitting the user to do arbitrary editing before resuming that command.

The commands available during recursive editing are the same ones available in the top-level editing loop and defined in the keymaps. Only a few special commands exit the recursive editing level; the others return to the recursive editing level when they finish. (The special commands for exiting are always available, but they do nothing when recursive editing is not in progress.)

All command loops, including recursive ones, set up all-purpose error handlers so that an error in a command run from the command loop will not exit the loop.

Minibuffer input is a special kind of recursive editing. It has a few special wrinkles, such as enabling display of the minibuffer and the minibuffer window, but fewer than you might suppose. Certain keys behave differently in the minibuffer, but that is only because of the minibuffer's local map; if you switch windows, you get the usual Emacs commands.

To invoke a recursive editing level, call the function `recursive-edit`. This function contains the command loop; it also contains a call to `catch` with tag `exit`, which makes it possible to exit the recursive editing level by throwing to `exit` (see Section 10.5.1 [Catch and Throw], page 126). 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-]* (`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` [Command]

This function invokes the editor command loop. It is called automatically by the initialization of Emacs, to let the user begin editing. When called from a Lisp program, it enters a recursive editing level.

If the current buffer is not the same as the selected window's buffer, `recursive-edit` saves and restores the current buffer. Otherwise, if you switch buffers, the buffer you switched to is current after `recursive-edit` returns.

In the following example, the function `simple-rec` first advances point one word, then enters a recursive edit, printing out a message in the echo area. The user can then do any editing desired, and then type *C-M-c* to exit and continue executing `simple-rec`.

```
(defun simple-rec ()
  (forward-word 1)
  (message "Recursive edit in progress")
  (recursive-edit)
  (forward-word 1))
     ⇒ simple-rec
(simple-rec)
     ⇒ nil
```

`exit-recursive-edit` [Command]

This function exits from the innermost recursive edit (including minibuffer input). Its definition is effectively `(throw 'exit nil)`.

`abort-recursive-edit` [Command]

This function aborts the command that requested the innermost recursive edit (including minibuffer input), by signaling `quit` after exiting the recursive edit. Its definition is effectively `(throw 'exit t)`. See Section 21.11 [Quitting], page 351.

`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.

21.14 Disabling Commands

Disabling a command marks the command as requiring user confirmation before it can be executed. Disabling is used for commands which might be confusing to beginning users, to prevent them from using the commands by accident.

The low-level mechanism for disabling a command is to put a non-`nil` `disabled` property on the Lisp symbol for the command. These properties are normally set up by the user's init file (see Section 39.1.2 [Init File], page 389, vol. 2) 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 389, vol. 2) 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.

21.15 Command History

The command loop keeps a history of the complex commands that have been executed, to make it convenient to repeat these commands. A *complex command* is one for which the interactive argument reading uses the minibuffer. This includes any *M-x* command, any *M-:* command, and any command whose `interactive` specification reads an argument from the minibuffer. Explicit use of the minibuffer during the execution of the command itself does not cause the command to be considered complex.

`command-history` [Variable]

This variable's value is a list of recent complex commands, each represented as a form to evaluate. It continues to accumulate all complex commands for the duration of the editing session, but when it reaches the maximum size (see Section 20.4 [Minibuffer History], page 289), 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 289), 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.

21.16 Keyboard Macros

A *keyboard macro* is a canned sequence of input events that can be considered a command and made the definition of a key. The Lisp representation of a keyboard macro is a string or vector containing the events. Don't confuse keyboard macros with Lisp macros (see Chapter 13 [Macros], page 181).

`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 21.8.2 [Reading One Event], page 344, 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 Terminals], page 67, vol. 2.

`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 Terminals], page 67, vol. 2.

`kbd-macro-termination-hook` [Variable]

This normal hook is run when a keyboard macro terminates, regardless of what caused it to terminate (reaching the macro end or an error which ended the macro prematurely).

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 21.7 [Input Events], page 327). The Emacs Lisp representation for a key sequence is a string or vector. Unless otherwise stated, any Emacs Lisp function that accepts a key sequence as an argument can handle both representations.

In the string representation, alphanumeric characters ordinarily stand for themselves; for example, `"a"` represents `a` and `"2"` represents *2*. Control character events are prefixed by the substring `"\C-"`, and meta characters by `"\M-"`; for example, `"\C-x"` represents the key *C-x*. In addition, the `TAB`, `RET`, `ESC`, and `DEL` events are represented by `"\t"`, `"\r"`, `"\e"`, and `"\d"` respectively. The string representation of a complete key sequence is the concatenation of the string representations of the constituent events; thus, `"\C-xl"` represents the key sequence *C-x l*.

Key sequences containing function keys, mouse button events, 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 21.7 [Input Events], page 327. For example, the vector `[?\C-x ?l]` represents the key sequence *C-x l*.

For examples of key sequences written in string and vector representations, Section "Init Rebinding" in *The GNU Emacs Manual*.

`kbd` *keyseq-text* [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") ⇒ "\C-x"
(kbd "C-x C-f") ⇒ "\C-x\C-f"
(kbd "C-x 4 C-f") ⇒ "\C-x4\C-f"
(kbd "X") ⇒ "X"
(kbd "RET") ⇒ "\^M"
(kbd "C-c SPC") ⇒ "\C-c "
(kbd "<f1> SPC") ⇒ [f1 32]
(kbd "C-M-<down>") ⇒ [C-M-down]
```

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 365, 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 367, 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 21.7.12 [Classifying Events], page 336. In this kind of binding, *binding* is a command.

`(`*`type item-name`*` . `*`binding`*`)`
: This specifies a binding which is also a simple menu item that displays as *item-name* in the menu. See Section 22.17.1.1 [Simple Menu Items], page 384.

`(`*`type item-name help-string`*` . `*`binding`*`)`
: This is a simple menu item with help string *help-string*.

`(`*`type`*` menu-item . `*`details`*`)`
: This specifies a binding which is also an extended menu item. This allows use of other features. See Section 22.17.1.2 [Extended Menu Items], page 385.

`(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 384.

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 365).

This conversion applies only to characters, not to function keys or other input events; thus, *`M-end`* has nothing to do with *`ESC end`*.

Here as an example is the local keymap for Lisp mode, a sparse keymap. It defines bindings for `DEL`, *`C-c C-z`*, *`C-M-q`*, and *`C-M-x`* (the actual value also contains a menu binding, which is omitted here for the sake of brevity).

```
lisp-mode-map
⇒
(keymap
 (3 keymap
    ;; C-c C-z
    (26 . run-lisp))
```

```
        (27 keymap
            ;; C-M-x, treated as ESC C-x
            (24 . lisp-send-defun))
        ;; This part is inherited from lisp-mode-shared-map.
        keymap
        ;; DEL
        (127 . backward-delete-char-untabify)
        (27 keymap
            ;; C-M-q, treated as ESC C-q
            (17 . indent-sexp)))
```

`keymapp` *object* [Function]

This function returns `t` if *object* is a keymap, `nil` otherwise. More precisely, this function tests for a list whose CAR is `keymap`, or for a symbol whose function definition satisfies `keymapp`.

```
(keymapp '(keymap))
    ⇒ t
(fset 'foo '(keymap))
(keymapp 'foo)
    ⇒ t
(keymapp (current-global-map))
    ⇒ t
```

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 384). 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 92) 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.

Sometimes you want to make a keymap that inherits from more than one map. You can use the function `make-composed-keymap` for this.

`make-composed-keymap` *maps* **&optional** *parent* [Function]

This function returns a new keymap composed of the existing keymap(s) *maps*, and optionally inheriting from a parent keymap *parent*. *maps* can be a single keymap or a list of more than one. When looking up a key in the resulting new map, Emacs searches in each of the *maps* in turn, and then in *parent*, stopping at the first match. A `nil` binding in any one of *maps* overrides any binding in *parent*, but it does not override any non-`nil` binding in any other of the *maps*.

For example, here is how Emacs sets the parent of `help-mode-map`, such that it inherits from both `button-buffer-map` and `special-mode-map`:

```
(defvar help-mode-map
  (let ((map (make-sparse-keymap)))
    (set-keymap-parent map
      (make-composed-keymap button-buffer-map special-mode-map))
    ... map) ... )
```

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.
- `goto-map` is the global keymap used for the *M-g* prefix key.
- `search-map` is the global keymap used for the *M-s* prefix key.
- `facemenu-keymap` is the global keymap used for the *M-o* prefix key.
- The other Emacs prefix keys are *C-x @*, *C-x a i*, *C-x ESC* and *ESC ESC*. They use keymaps that have no special names.

The keymap binding of a prefix key is used for looking up the event that follows the prefix key. (It may instead be a symbol whose function definition is a keymap. The effect is the same, but the symbol serves as a name for the prefix key.) Thus, the binding of *C-x* is the symbol `Control-X-prefix`, whose function cell holds the keymap for *C-x* commands. (The same keymap is also the value of `ctl-x-map`.)

Prefix key definitions can appear in any active keymap. The definitions of *C-c*, *C-x*, *C-h* and `ESC` as prefix keys appear in the global map, so these prefix keys are always available. Major and minor modes can redefine a key as a prefix by putting a prefix key definition for it in the local map or the minor mode's map. See Section 22.7 [Active Keymaps], page 367.

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 384).

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 368, 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 162, vol. 2), the non-`nil` map properties of the string override those of the buffer (if the underlying buffer text contains map properties in its text properties or overlays, they are ignored).

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 162, vol. 2.

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 284.

Emacs has other keymaps that are used in a different way—translating events within `read-key-sequence`. See Section 22.14 [Translation Keymaps], page 378.

See Appendix G [Standard Keymaps], page 481, vol. 2, for a list of some standard keymaps.

`current-active-maps` **&optional** *olp position* [Function]

This returns the list of active keymaps that would be used by the command loop in the current circumstances to look up a key sequence. Normally it ignores `overriding-local-map` and `overriding-terminal-local-map`, but if *olp* is non-`nil` then it pays attention to them. *position* can optionally be either an event position as returned by `event-start` or a buffer position, and may change the keymaps as described for `key-binding`.

`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 373).

When commands are remapped (see Section 22.13 [Remapping Commands], page 378), `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 378) Emacs looks for them in the active keymaps. Here is a pseudo-Lisp description of the order and conditions for searching them:

```
(or (cond
     (overriding-terminal-local-map
      (find-in overriding-terminal-local-map))
     (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)))
```

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 371.) 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.

When a match is found (see Section 22.10 [Key Lookup], page 371), 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.

The function finally found might also be remapped. See Section 22.13 [Remapping Commands], page 378.

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. The return value is a reference, not a copy; if you use `define-key` or other functions on it you will alter global bindings.

```
(current-global-map)
⇒ (keymap [set-mark-command beginning-of-line ...
            delete-backward-char])
```

`current-local-map` [Function]

This function returns the current buffer's local keymap, or `nil` if it has none. In the following example, the keymap for the '`*scratch*`' buffer (using Lisp Interaction mode) is a sparse keymap in which the entry for `ESC`, ASCII code 27, is another sparse keymap.

```
(current-local-map)
⇒ (keymap
    (10 . eval-print-last-sexp)
    (9 . lisp-indent-line)
    (127 . backward-delete-char-untabify)
    (27 keymap
        (24 . eval-defun)
        (17 . indent-sexp)))
```

`current-local-map` returns a reference to the local keymap, not a copy of it; if you use `define-key` or other functions on it you will alter local bindings.

`current-minor-mode-maps` [Function]

This function returns a list of the keymaps of currently enabled minor modes.

`use-global-map` *keymap* [Function]

This function makes *keymap* the new current global keymap. It returns `nil`.

It is very unusual to change the global keymap.

`use-local-map` *keymap* [Function]

This function makes *keymap* the new local keymap of the current buffer. If *keymap* is `nil`, then the buffer has no local keymap. `use-local-map` returns `nil`. Most major mode commands use this function.

`minor-mode-map-alist` [Variable]

This variable is an alist describing keymaps that may or may not be active according to the values of certain variables. Its elements look like this:

```
(variable . keymap)
```

The keymap *keymap* is active whenever *variable* has a non-`nil` value. Typically *variable* is the variable that enables or disables a minor mode. See Section 23.3.2 [Keymaps and Minor Modes], page 415.

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 415, for more information about minor modes. See also `minor-mode-key-binding` (see Section 22.11 [Functions for Key Lookup], page 373).

`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 Terminals], page 67, vol. 2. It is used to implement incremental search mode.

`overriding-local-map-menu-flag` [Variable]

If this variable is non-`nil`, the value of `overriding-local-map` or `overriding-terminal-local-map` can affect the display of the menu bar. The default value is `nil`, so those map variables have no effect on the menu bar.

Note that these two map variables do affect the execution of key sequences entered using the menu bar, even if they do not affect the menu bar display. So if a menu bar key sequence comes in, you should clear the variables before looking up and executing that key sequence. Modes that use the variables would typically do this anyway; normally they respond to events that they do not handle by "unreading" them and exiting.

`special-event-map` [Variable]

This variable holds a keymap for special events. If an event type has a binding in this keymap, then it is special, and the binding for the event is run directly by `read-event`. See Section 21.9 [Special Events], page 350.

`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 21.7 [Input Events], page 327. 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:

`nil`
: `nil` means that the events used so far in the lookup form an undefined key. When a keymap fails to mention an event type at all, and has no default binding, that is equivalent to a binding of `nil` for that event type.

command
: The events used so far in the lookup form a complete key, and *command* is its binding. See Section 12.1 [What Is a Function], page 163.

array
: The array (either a string or a vector) is a keyboard macro. The events used so far in the lookup form a complete key, and the array is its binding. See Section 21.16 [Keyboard Macros], page 358, for more information.

keymap
: The events used so far in the lookup form a prefix key. The next event of the key sequence is looked up in *keymap*.

list
: The meaning of a list depends on what it contains:

  - If the CAR of *list* is the symbol `keymap`, then the list is a keymap, and is treated as a keymap (see above).
  - If the CAR of *list* is `lambda`, then the list is a lambda expression. This is presumed to be a function, and is treated as such (see above). In order to execute properly as a key binding, this function must be a command—it must have an `interactive` specification. See Section 21.2 [Defining Commands], page 316.
  - 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 321).

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`.

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 361.)

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 21.8.1 [Key Sequence Input], page 342). 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` [User Option]

This variable is the meta-prefix character code. It is used for translating a meta character to a two-character sequence so it can be looked up in a keymap. For useful results, the value should be a prefix event (see Section 22.6 [Prefix Keys], page 365). 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 380). You can also use `define-key`, a more general function; then you must explicitly specify the map to change.

When choosing the key sequences for Lisp programs to rebind, please follow the Emacs conventions for use of various keys (see Section D.2 [Key Binding Conventions], page 446, vol. 2).

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 360) 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 371.) 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 384).

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 378).

`substitute-key-definition` *olddef newdef keymap* **&optional** *oldmap* [Function]

This function replaces *olddef* with *newdef* for any keys in *keymap* that were bound to *olddef*. In other words, *olddef* is replaced with *newdef* wherever it appears. The function returns `nil`.

For example, this redefines *C-x C-f*, if you do it in an Emacs with standard bindings:

```
(substitute-key-definition
 'find-file 'find-file-read-only (current-global-map))
```

If *oldmap* is non-`nil`, that changes the behavior of `substitute-key-definition`: the bindings in *oldmap* determine which keys to rebind. The rebindings still happen in *keymap*, not in *oldmap*. Thus, you can change one map under the control of the bindings in another. For example,

```
(substitute-key-definition
  'delete-backward-char 'my-funny-delete
  my-map global-map)
```

puts the special deletion command in `my-map` for whichever keys are globally bound to the standard deletion command.

Here is an example showing a keymap before and after substitution:

```
(setq map '(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 378). 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 9, vol. 2).

Since this function modifies *keymap*, you would normally use it on a newly created keymap. Operating on an existing keymap that is used for some other purpose is likely to cause trouble; for example, suppressing `global-map` would make it impossible to use most of Emacs.

This function can be used to initialize the local keymap of a major mode for which insertion of text is not desirable. But usually such a mode should be derived from `special-mode` (see Section 23.2.5 [Basic Major Modes], page 407); then its keymap will automatically inherit from `special-mode-map`, which is already suppressed. Here is how `special-mode-map` is defined:

```
(defvar special-mode-map
  (let ((map (make-sparse-keymap)))
    (suppress-keymap map)
    (define-key map "q" 'quit-window)
    ...
    map))
```

22.13 Remapping Commands

A special kind of key binding can be used to *remap* one command to another, without having to refer to the key sequence(s) bound to the original command. To use this feature, make a key binding for a key sequence that starts with the dummy event `remap`, followed by the command name you want to remap; for the binding, specify the new definition (usually a command name, but possibly any other valid definition for a key binding).

For example, suppose My mode provides a special command `my-kill-line`, which should be invoked instead of `kill-line`. To establish this, its mode keymap should contain the following remapping:

```
(define-key my-mode-map [remap kill-line] 'my-kill-line)
```

Then, whenever `my-mode-map` is active, if the user types `C-k` (the default global key sequence for `kill-line`) Emacs will instead run `my-kill-line`.

Note that remapping only takes place through active keymaps; for example, putting a remapping in a prefix keymap like `ctl-x-map` typically has no effect, as such keymaps are not themselves active. In addition, remapping only works through a single level; in the following example,

```
(define-key my-mode-map [remap kill-line] 'my-kill-line)
(define-key my-mode-map [remap my-kill-line] 'my-other-kill-line)
```

`kill-line` is *not* remapped to `my-other-kill-line`. Instead, if an ordinary key binding specifies `kill-line`, it is remapped to `my-kill-line`; if an ordinary binding specifies `my-kill-line`, it is remapped to `my-other-kill-line`.

To undo the remapping of a command, remap it to `nil`; e.g.

```
(define-key my-mode-map [remap kill-line] nil)
```

`command-remapping` *command* **&optional** *position keymaps* [Function]

This function returns the remapping for *command* (a symbol), given the current active keymaps. If *command* is not remapped (which is the usual situation), or not a symbol, the function returns `nil`. `position` can optionally specify a buffer position or an event position to determine the keymaps to use, as in `key-binding`.

If the optional argument `keymaps` is non-`nil`, it specifies a list of keymaps to search in. This argument is ignored if `position` is non-`nil`.

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 `input-decode-map`, then `local-function-key-map`, and then against `key-translation-map`.

`input-decode-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 `input-decode-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 `input-decode-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]`.

The value of `input-decode-map` is usually set up automatically according to the terminal’s Terminfo or Termcap entry, but sometimes those need help from terminal-specific Lisp files. Emacs comes with terminal-specific files for many common terminals; their main purpose is to make entries in `input-decode-map` beyond those that can be deduced from Termcap and Terminfo. See Section 39.1.3 [Terminal-Specific], page 390, vol. 2.

`local-function-key-map` [Variable]

This variable holds a keymap similar to `input-decode-map` except that it describes key sequences which should be translated to alternative interpretations that are usually preferred. It applies after `input-decode-map` and before `key-translation-map`.

Entries in `local-function-key-map` are ignored if they conflict with bindings made in the minor mode, local, or global keymaps. I.e. the remapping only applies if the original key sequence would otherwise not have any binding.

`local-function-key-map` inherits from `function-key-map`, but the latter should not be used directly.

`key-translation-map` [Variable]

This variable is another keymap used just like `input-decode-map` to translate input events into other events. It differs from `input-decode-map` in that it goes to work after `local-function-key-map` is finished rather than before; it receives the results of translation by `local-function-key-map`.

Just like `input-decode-map`, but unlike `local-function-key-map`, this keymap is applied regardless of whether the input key-sequence has a normal binding. Note however that actual key bindings can have an effect on `key-translation-map`, even though they are overridden by it. Indeed, actual key bindings override `local-function-key-map` and thus may alter the key sequence that `key-translation-map` receives. Clearly, it is better to avoid this type of situation.

The intent of `key-translation-map` is for users to map one character set to another, including ordinary characters normally bound to `self-insert-command`.

You can use `input-decode-map`, `local-function-key-map`, and `key-translation-map` for more than simple aliases, by using a function, instead of a key sequence, as the "translation" of a key. Then this function is called to compute the translation of that key.

The key translation function receives one argument, which is the prompt that was specified in `read-key-sequence`—or `nil` if the key sequence is being read by the editor command loop. In most cases you can ignore the prompt value.

If the function reads input itself, it can have the effect of altering the event that follows. For example, here's how to define *C-c h* to turn the character that follows into a Hyper character:

```
(defun hyperify (prompt)
  (let ((e (read-event)))
    (vector (if (numberp e)
                (logior (lsh 1 24) e)
              (if (memq 'hyper (event-modifiers e))
                  e
                (add-event-modifier "H-" e))))))

(defun add-event-modifier (string e)
  (let ((symbol (if (symbolp e) e (car e))))
    (setq symbol (intern (concat string
                                 (symbol-name symbol))))
    (if (symbolp e)
        symbol
      (cons symbol (cdr e)))))

(define-key local-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 33.9.8 [Terminal I/O Encoding], page 205, vol. 2. 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 389, vol. 2) 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 213), 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 246, not the byte code 246 (*M-v*) sent by a Latin-1 terminal. In order to use this binding, you need to teach Emacs how to decode the keyboard by using an appropriate input method (see Section "Input Methods" in *The GNU Emacs Manual*).

`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-l")
    ⇒ nil
(global-set-key "\C-l\C-l" '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 no-remap* [Function]

This function is a subroutine used by the `where-is` command (see Section "Help" in *The GNU Emacs Manual*). It returns a list of all key sequences (of any length) that are bound to *command* in a set of keymaps.

The argument *command* can be any object; it is compared with all keymap entries using `eq`.

If *keymap* is `nil`, then the maps used are the current active keymaps, disregarding `overriding-local-map` (that is, pretending its value is `nil`). If *keymap* is a keymap, then the maps searched are *keymap* and the global keymap. If *keymap* is a list of keymaps, only those keymaps are searched.

Usually it's best to use `overriding-local-map` as the expression for *keymap*. Then `where-is-internal` searches precisely the keymaps that are active. To search only the global map, pass the value `(keymap)` (an empty keymap) as *keymap*.

If *firstonly* is `non-ascii`, then the value is a single vector representing the first key sequence found, rather than a list of all possible key sequences. If *firstonly* is `t`, then the value is the first key sequence, except that key sequences consisting entirely of ASCII characters (or meta variants of ASCII characters) are preferred to all other key sequences and that the return value can never be a menu binding.

If *noindirect* is non-`nil`, `where-is-internal` doesn't follow indirect keymap bindings. This makes it possible to search for an indirect definition itself.

The fifth argument, *no-remap*, determines how this function treats command remappings (see Section 22.13 [Remapping Commands], page 378). There are two cases of interest:

If a command *other-command* is remapped to *command*:

If *no-remap* is `nil`, find the bindings for *other-command* and treat them as though they are also bindings for *command*. If *no-remap* is non-`nil`, include the vector `[remap` ***`other-command`***`]` in the list of possible key sequences, instead of finding those bindings.

If *command* is remapped to *other-command*:

If *no-remap* is `nil`, return the bindings for *other-command* rather than *command*. If *no-remap* is non-`nil`, return the bindings for *command*, ignoring the fact that it is remapped.

`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 361.) The string should describe the purpose of the menu's commands. Emacs displays the overall prompt string as the menu title in some cases, depending on the toolkit (if any) used for displaying menus.[1] Keyboard menus also display the overall prompt string.

The easiest way to construct a keymap with a prompt string is to specify the string as an argument when you call `make-keymap`, `make-sparse-keymap` (see Section 22.4 [Creating Keymaps], page 363), or `define-prefix-command` (see [Definition of define-prefix-command], page 366). 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 395).

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:

[1] It is required for menus which do not use a toolkit, e.g. under MS-DOS.

```
(item-string . real-binding)
```

The CAR, *item-string*, is the string to be displayed in the menu. It should be short—preferably one to three words. It should describe the action of the command it corresponds to. Note that not all graphical toolkits can display non-ASCII text in menus (it will work for keyboard menus and will work to a large extent with the GTK+ toolkit).

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 167, vol. 2).

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 419).

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 387.

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.

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 167, vol. 2). 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 separators consisting of dashes only.

`"--shadow-etched-out"`
: A single line with a 3D raised appearance.

`"--shadow-etched-in-dash"`
: A single dashed line with a 3D sunken appearance.

`"--shadow-etched-out-dash"`
: A single dashed line with a 3D raised appearance.

`"--shadow-double-etched-in"`
: Two lines with a 3D sunken appearance.

`"--shadow-double-etched-out"`
: Two lines with a 3D raised appearance.

`"--shadow-double-etched-in-dash"`
: Two dashed lines with a 3D sunken appearance.

`"--shadow-double-etched-out-dash"`
: Two dashed lines with a 3D raised appearance.

You can also give these names in another style, adding a colon after the double-dash and replacing each single dash with capitalization of the following word. Thus, `"--:singleLine"`, is equivalent to `"--single-line"`.

You can use a longer form to specify keywords such as `:enable` and `:visible` for a menu separator:

(menu-item *separator-type* nil . *item-property-list*)

For example:

```
(menu-item "--" nil :visible (boundp 'foo))
```

Some systems and display toolkits don't really handle all of these separator types. If you use a type that isn't supported, the menu displays a similar kind of separator that is supported.

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`.

22.17.1.5 Toolkit Differences

The various toolkits with which you can build Emacs do not all support the same set of features for menus. Some code works as expected with one toolkit, but not under another.

One example is menu actions or buttons in a top-level menu bar. The following works with the Lucid toolkit or on MS Windows, but not with GTK or Nextstep, where clicking on the item has no effect.

```
(defun menu-action-greet ()
   (interactive)
   (message "Hello Emacs User!"))

(defun top-level-menu ()
  (interactive)
  (define-key lisp-interaction-mode-map [menu-bar m]
     '(menu-item "Action Button" menu-action-greet)))
```

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 88, vol. 2.)

If the prefix key ends with a mouse event, Emacs handles the menu keymap by popping up a visible menu, so that the user can select a choice with the mouse. When the user clicks on a menu item, the event generated is whatever character or symbol has the binding that brought about that menu item. (A menu item may generate a series of events if the menu has multiple levels or comes from the menu bar.)

It's often best to use a button-down event to trigger the menu. Then the user can select a menu item by releasing the button.

If the menu keymap contains a binding to a nested keymap, the nested keymap specifies a *submenu*. There will be a menu item, labeled by the nested keymap's item string, and clicking on this item automatically pops up the specified submenu. As a special exception, if the menu keymap contains a single nested keymap and no other menu items, the menu shows the contents of the nested keymap directly, not as a submenu.

However, if Emacs is compiled without X toolkit support, submenus are not supported. Each nested keymap is shown as a menu item, but clicking on it does not automatically pop up the submenu. If you wish to imitate the effect of submenus, you can do that by giving a nested keymap an item string which starts with '`@`'. This causes Emacs to display the nested keymap using a separate *menu pane*; the rest of the item string after the '`@`' is the pane label. If Emacs is compiled without X toolkit support, menu panes are not used; in that case, a '`@`' at the beginning of an item string is omitted when the menu label is displayed, and has no other effect.

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.

`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 385). 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

On graphical displays, there is usually a *menu bar* at the top of each frame. See Section "Menu Bars" in *The GNU Emacs Manual*. Menu bar items are subcommands of the fake "function key" `menu-bar`, as defined in the active keymaps.

To add an item to the menu bar, invent a fake "function key" of your own (let's call it *key*), and make a binding for the key sequence `[menu-bar key]`. Most often, the binding is a menu keymap, so that pressing a button on the menu bar item leads to another menu.

When more than one active keymap defines the same "function key" for the menu bar, the item appears just once. If the user clicks on that menu bar item, it brings up a single, combined menu containing all the subcommands of that item—the global subcommands, the local subcommands, and the minor mode subcommands.

The variable `overriding-local-map` is normally ignored when determining the menu bar contents. That is, the menu bar is computed from the keymaps that would be active if `overriding-local-map` were `nil`. See Section 22.7 [Active Keymaps], page 367.

Here's an example of setting up a menu bar item:

```
;; Make a menu keymap (with a prompt string)
;; and make it the menu bar item's definition.
(define-key global-map [menu-bar words]
  (cons "Words" (make-sparse-keymap "Words")))

;; Define specific subcommands in this menu.
(define-key global-map
  [menu-bar words forward]
  '("Forward word" . forward-word))
(define-key global-map
  [menu-bar words backward]
  '("Backward word" . backward-word))
```

A local keymap can cancel a menu bar item made by the global keymap by rebinding the same fake function key with `undefined` as the binding. For example, this is how Dired suppresses the '`Edit`' menu bar item:

```
(define-key dired-mode-map [menu-bar edit] 'undefined)
```

Here, `edit` is the 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.

Next to every menu bar item, Emacs displays a key binding that runs the same command (if such a key binding exists). This serves as a convenient hint for users who do not know the key binding. If a command has multiple bindings, Emacs normally displays the first one it finds. You can specify one particular key binding by assigning an `:advertised-binding` symbol property to the command. See Section 24.3 [Keys in Documentation], page 454.

22.17.6 Tool bars

A *tool bar* is a row of clickable icons at the top of a frame, just below the menu bar. See Section "Tool Bars" in *The GNU Emacs Manual*.

On each frame, the frame parameter `tool-bar-lines` controls how many lines' worth of height to reserve for the tool bar. A zero value suppresses the tool bar. If the value is nonzero, and `auto-resize-tool-bars` is non-`nil`, the tool bar expands and contracts automatically as needed to hold the specified contents. If the value is `grow-only`, the tool bar expands automatically, but does not contract automatically.

The tool bar contents are controlled by a menu keymap attached to a fake "function key" called `tool-bar` (much like the way the menu bar is controlled). So you define a tool bar item using `define-key`, like this:

```
(define-key global-map [tool-bar key] item)
```

where *key* is a fake "function key" to distinguish this item from other items, and *item* is a menu item key binding (see Section 22.17.1.2 [Extended Menu Items], page 385), 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 167, vol. 2).

In addition, you should use the `:image` property; this is how you specify the image to display in the tool bar:

`:image` *`image`*

images 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 `:rtl` property specifies an alternative image to use for right-to-left languages. Only the GTK+ version of Emacs supports this at present.

Like the menu bar, the tool bar can display separators (see Section 22.17.1.3 [Menu Separators], page 387). Tool bar separators are vertical rather than horizontal, though, and only a single style is supported. They are represented in the tool bar keymap by `(menu-item "--")` entries; properties like `:visible` are not supported for tool bar separators. Separators are rendered natively in GTK+ and Nextstep tool bars; in the other cases, they are rendered using an image of a vertical line.

The default tool bar is defined so that items specific to editing do not appear for major modes whose command symbol has a `mode-class` property of `special` (see Section 23.2.1 [Major Mode Conventions], page 399). 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 ,(purecopy "tool bar") ignore
    :filter tool-bar-make-keymap))
```

The function `tool-bar-make-keymap`, in turn, derives the actual tool bar map dynamically from the value of the variable `tool-bar-map`. Hence, you should normally adjust the default (global) tool bar by changing that map. Some major modes, such as Info mode, completely replace the global tool bar by making `tool-bar-map` buffer-local and setting it to a different keymap.

There are two convenience functions for defining tool bar items, as follows.

`tool-bar-add-item` *icon def key* **&rest** *props* [Function]

This function adds an item to the tool bar by modifying `tool-bar-map`. The image to use is defined by *icon*, which is the base name of an XPM, XBM or PBM image file to be located by `find-image`. Given a value ‘`"exit"`’, say, ‘`exit.xpm`’, ‘`exit.pbm`’ and ‘`exit.xbm`’ would be searched for in that order on a color display. On a monochrome display, the search order is ‘`.pbm`’, ‘`.xbm`’ and ‘`.xpm`’. The binding to use is the command *def*, and *key* is the fake function key symbol in the prefix keymap. The remaining arguments *props* are additional property list elements to add to the menu item specification.

To define items in some local map, bind `tool-bar-map` with `let` around calls of this function:

```
(defvar foo-tool-bar-map
  (let ((tool-bar-map (make-sparse-keymap)))
    (tool-bar-add-item ...)
    ...
    tool-bar-map))
```

`tool-bar-add-item-from-menu` *command icon* **&optional** *map* **&rest** *props* [Function]

This function is a convenience for defining tool bar items which are consistent with existing menu bar bindings. The binding of *command* is looked up in the menu bar in *map* (default `global-map`) and modified to add an image specification for *icon*, which is found in the same way as by `tool-bar-add-item`. The resulting binding is then placed in `tool-bar-map`, so use this function only for global tool bar items.

map must contain an appropriate keymap bound to `[menu-bar]`. The remaining arguments *props* are additional property list elements to add to the menu item specification.

`tool-bar-local-item-from-menu` *command icon in-map* **&optional** *from-map* **&rest** *props* [Function]

This function is used for making non-global tool bar items. Use it like `tool-bar-add-item-from-menu` except that *in-map* specifies the local map to make the definition in. The argument *from-map* is like the *map* argument of `tool-bar-add-item-from-menu`.

`auto-resize-tool-bars` [Variable]

If this variable is non-`nil`, the tool bar automatically resizes to show all defined tool bar items—but not larger than a quarter of the frame's height.

If the value is `grow-only`, the tool bar expands automatically, but does not contract automatically. To contract the tool bar, the user has to redraw the frame by entering *`C-l`*.

If Emacs is built with GTK or Nextstep, the tool bar can only show one line, so this variable has no effect.

`auto-raise-tool-bar-buttons` [Variable]

If this variable is non-`nil`, tool bar items display in raised form when the mouse moves over them.

`tool-bar-button-margin` [Variable]

This variable specifies an extra margin to add around tool bar items. The value is an integer, a number of pixels. The default is 4.

`tool-bar-button-relief` [Variable]

This variable specifies the shadow width for tool bar items. The value is an integer, a number of pixels. The default is 1.

`tool-bar-border` [Variable]

This variable specifies the height of the border drawn below the tool bar area. An integer 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 21.7.2 [Function Keys], page 328, 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 360, and Chapter 35 [Syntax Tables], page 234, vol. 2.

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 389, vol. 2), but Lisp programs can set them also. See Appendix H [Standard Hooks], page 484, vol. 2, for a list of some standard hook variables.

Most of the hooks in Emacs are *normal hooks*. These variables contain lists of functions to be called with no arguments. By convention, whenever the hook name ends in '`-hook`', that tells you it is normal. We try to make all hooks normal, as much as possible, so that you can use them in a uniform way.

Every major mode command is supposed to run a normal hook called the *mode hook* as one of the last steps of initialization. This makes it easy for a user to customize the behavior of the mode, by overriding the buffer-local variable assignments already made by the mode. Most minor mode functions also run a mode hook at the end. But hooks are used in other contexts too. For example, the hook `suspend-hook` runs just before Emacs suspends itself (see Section 39.2.2 [Suspending Emacs], page 393, vol. 2).

The recommended way to add a hook function to a hook is by calling `add-hook` (see Section 23.1.2 [Setting Hooks], page 398). The hook functions may be any of the valid kinds of functions that `funcall` accepts (see Section 12.1 [What Is a Function], page 163). 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.

23.1.1 Running Hooks

In this section, we document the `run-hooks` function, which is used to run a normal hook. We also document the functions for running various kinds of abnormal hooks.

`run-hooks` **&rest** *hookvars* [Function]

This function takes one or more normal hook variable names as arguments, and runs each hook in turn. Each argument should be a symbol that is a normal hook variable. These arguments are processed in the order specified.

If a hook variable has a non-`nil` value, that value should be a list of functions. `run-hooks` calls all the functions, one by one, with no arguments.

The hook variable's value can also be a single function—either a lambda expression or a symbol with a function definition—which `run-hooks` calls. But this usage is obsolete.

If the hook variable is buffer-local, the buffer-local variable will be used instead of the global variable. However, if the buffer-local variable contains the element `t`, the global hook variable will be run as well.

`run-hook-with-args` *hook* **&rest** *args* [Function]

This function runs an abnormal hook by calling all the hook functions in *hook*, passing each one the arguments *args*.

`run-hook-with-args-until-failure` *hook* **&rest** *args* [Function]

This function runs an abnormal hook by calling each hook function in turn, stopping if one of them "fails" by returning `nil`. Each hook function is passed the arguments *args*. If this function stops because one of the hook functions fails, it returns `nil`; otherwise it returns a non-`nil` value.

`run-hook-with-args-until-success` *hook* **&rest** *args* [Function]

This function runs an abnormal hook by calling each hook function, stopping if one of them "succeeds" by returning a non-`nil` value. Each hook function is passed the arguments *args*. If this function stops because one of the hook functions returns a non-`nil` value, it returns that value; otherwise it returns `nil`.

`with-wrapper-hook` *hook args* **&rest** *body* [Macro]

This macro runs the abnormal hook `hook` as a series of nested "wrapper functions" around the *body* forms. The effect is similar to nested `around` advices (see Section 17.3 [Around-Advice], page 236).

Each hook function should accept an argument list consisting of a function *fun*, followed by the additional arguments listed in *args*. The first hook function is passed a function *fun* that, if it is called with arguments *args*, performs *body* (i.e., the default operation). The *fun* passed to each successive hook function is constructed from all the preceding hook functions (and *body*); if this *fun* is called with arguments *args*, it does what the `with-wrapper-hook` call would if the preceding hook functions were the only ones in *hook*.

Each hook function may call its *fun* argument as many times as it wishes, including never. In that case, such a hook function acts to replace the default definition altogether, and any preceding hook functions. Of course, a subsequent hook function may do the same thing.

Each hook function definition is used to construct the *fun* passed to the next hook function in *hook*, if any. The last or "outermost" *fun* is called once to produce the overall effect.

When might you want to use a wrapper hook? The function `filter-buffer-substring` illustrates a common case. There is a basic functionality, performed by *body*—in this case, to extract a buffer-substring. Then any number of hook functions can act in sequence to modify that string, before returning the final result. A wrapper-hook also allows for a hook function to completely replace the default definition (by not calling *fun*).

`run-hook-wrapped` *hook wrap-function* **&rest** *args* [Function]

This function is similar to `run-hook-with-args-until-success`. Like that function, it runs the functions on the abnormal hook `hook`, stopping at the first one that returns non-`nil`. Instead of calling the hook functions directly, though, it actually calls `wrap-function` with arguments `fun` and `args`.

23.1.2 Setting Hooks

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 'auto-fill-mode)
```

`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.

If *function* has a non-`nil` property `permanent-local-hook`, then `kill-all-local-variables` (or changing major modes) won't delete it from the hook variable's local value.

For a normal hook, hook functions should be designed so that the order in which they are executed does not matter. Any dependence on the order is asking for trouble. However, the order is predictable: normally, *function* goes at the front of the hook list, so it is executed first (barring another `add-hook` call). If the optional argument *append* is non-`nil`, the new hook function goes at the end of the hook list and is 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. 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 one major mode at a time. Every major mode is associated with a *major mode command*, whose name should end in '`-mode`'. This command takes care of switching to that mode in the current buffer, by setting various buffer-local variables such as a local keymap. See Section 23.2.1 [Major Mode Conventions], page 399.

The least specialized major mode is called *Fundamental mode*, which has no mode-specific definitions or variable settings.

`fundamental-mode` [Command]

This is the major mode command for Fundamental mode. Unlike other mode commands, it does *not* run any mode hooks (see Section 23.2.1 [Major Mode Conventions], page 399), since you are not supposed to customize this mode.

The easiest way to write a major mode is to use the macro `define-derived-mode`, which sets up the new mode as a variant of an existing major mode. See Section 23.2.4 [Derived Modes], page 405. We recommend using `define-derived-mode` even if the new mode is not an obvious derivative of another mode, as it automatically enforces many coding conventions for you. See Section 23.2.5 [Basic Major Modes], page 407, for common modes to derive from.

The standard GNU Emacs Lisp directory tree contains the code for several major modes, in files such as '`text-mode.el`', '`texinfo.el`', '`lisp-mode.el`', and '`rmail.el`'. You can study these libraries to see how modes are written.

`major-mode` [User Option]

The buffer-local value of this variable holds the symbol for the current major mode. Its default value holds the default major mode for new buffers. The standard default value is `fundamental-mode`.

If the default value is `nil`, then whenever Emacs creates a new buffer via a command such as *C-x b* (`switch-to-buffer`), the new buffer is put in the major mode of the previously current buffer. As an exception, if the major mode of the previous buffer has a `mode-class` symbol property with value `special`, the new buffer is put in Fundamental mode (see Section 23.2.1 [Major Mode Conventions], page 399).

23.2.1 Major Mode Conventions

The code for every major mode should follow various coding conventions, including conventions for local keymap and syntax table initialization, function and variable names, and hooks.

If you use the `define-derived-mode` macro, it will take care of many of these conventions automatically. See Section 23.2.4 [Derived Modes], page 405. Note also that Fundamental mode is an exception to many of these conventions, because it represents the default state of Emacs.

The following list of conventions is only partial. Each major mode should aim for consistency in general with other Emacs major modes, as this makes Emacs as a whole more coherent. It is impossible to list here all the possible points where this issue might come up; if the Emacs developers point out an area where your major mode deviates from the usual conventions, please make it compatible.

- Define a major mode command whose name ends in '`-mode`'. When called with no arguments, this command should switch to the new mode in the current buffer by setting up the keymap, syntax table, and buffer-local variables in an existing buffer. It should not change the buffer's contents.
- Write a documentation string for this command that describes the special commands available in this mode. See Section 23.2.3 [Mode Help], page 405.

  The documentation string may include the special documentation substrings, '`\[`*command*`]`', '`\{`*keymap*`}`', and '`\<`*keymap*`>`', which allow the help display to adapt automatically to the user's own key bindings. See Section 24.3 [Keys in Documentation], page 454.
- 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 152.
- The major mode command should set the variable `major-mode` to the major mode command symbol. This is how `describe-mode` discovers which documentation to print.
- The major mode command should set the variable `mode-name` to the "pretty" name of the mode, usually a string (but see Section 23.4.2 [Mode Line Data], page 419, for other possible forms). The name of the mode 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 444, vol. 2.
- In a major mode for editing some kind of structured text, such as a programming language, indentation of text according to structure is probably useful. So the mode should set `indent-line-function` to a suitable function, and probably customize other variables for indentation. See Section 23.7 [Auto-Indentation], page 440.
- 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 367, 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 142, for advice about how to write the code to set up the mode's keymap variable.
- The key sequences bound in a major mode keymap should usually start with *`C-c`*, followed by a control character, a digit, or *`{`*, *`}`*, *`<`*, *`>`*, *`:`* or *`;`*. The other punctuation characters are reserved for minor modes, and ordinary letters are reserved for users.

  A major mode can also rebind the keys *`M-n`*, *`M-p`* and *`M-s`*. The bindings for *`M-n`* and *`M-p`* should normally be some kind of "moving forward and backward", but this does not necessarily mean cursor motion.

It is legitimate for a major mode to rebind a standard key sequence if it provides a command that does "the same job" in a way better suited to the text this mode is used for. For example, a major mode for editing a programming language might redefine `C-M-a` to "move to the beginning of a function" in a way that works better for that language.

It is also legitimate for a major mode to rebind a standard key sequence whose standard meaning is rarely useful in that mode. For instance, minibuffer modes rebind `M-r`, whose standard meaning is rarely of any use in the minibuffer. Major modes such as Dired or Rmail that do not allow self-insertion of text can reasonably redefine letters and other printing characters as special commands.

- Major modes for editing text should not define `RET` to do anything other than insert a newline. However, it is ok for specialized modes for text that users don't directly edit, such as Dired and Info modes, to redefine `RET` to do something entirely different.
- Major modes should not alter options that are primarily a matter of user preference, such as whether Auto-Fill mode is enabled. Leave this to each user to decide. However, a major mode should customize other variables so that Auto-Fill mode will work usefully *if* the user decides to use it.
- The mode may have its own syntax table or may share one with other related modes. If it has its own syntax table, it should store this in a variable named `*modename*-mode-syntax-table`. See Chapter 35 [Syntax Tables], page 234, vol. 2.
- 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 36.2 [Defining Abbrevs], page 251, vol. 2.
- 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 429).
- Each face that the mode defines should, if possible, inherit from an existing Emacs face. See Section 38.12.8 [Basic Faces], page 336, vol. 2, and Section 23.6.7 [Faces for Font Lock], page 436.
- 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 427).
- The mode can specify a local value for `eldoc-documentation-function` to tell ElDoc mode how to handle this mode.
- The mode can specify how to complete various keywords by adding one or more buffer-local entries to the special hook `completion-at-point-functions`. See Section 20.6.8 [Completion in Buffers], page 306.

- 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 150.

  With rare exceptions, the only reasonable way to use `make-variable-buffer-local` in a Lisp package is for a variable which is used only within that package. Using it on a variable used by other packages would interfere with them.
- Each major mode should have a normal *mode hook* named `*modename*-mode-hook`. The very last thing the major mode command should do is to call `run-mode-hooks`. This runs the normal hook `change-major-mode-after-body-hook`, the mode hook, and then the normal hook `after-change-major-mode-hook`. See Section 23.2.6 [Mode Hooks], page 408.
- The major mode command may start by calling some other major mode command (called the *parent mode*) and then alter some of its settings. A mode that does this is called a *derived mode*. The recommended way to define one is to use the `define-derived-mode` macro, but this is not required. Such a mode should call the parent mode command inside a `delay-mode-hooks` form. (Using `define-derived-mode` does this automatically.) See Section 23.2.4 [Derived Modes], page 405, and Section 23.2.6 [Mode Hooks], page 408.
- 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 152).
- If this mode is appropriate only for specially-prepared text produced by the mode itself (rather than by the user typing at the keyboard or by an external file), then the major mode command symbol should have a property named `mode-class` with value `special`, put on as follows:

  ```
  (put 'funny-mode 'mode-class 'special)
  ```

  This tells Emacs that new buffers created while the current buffer is in Funny mode should not be put in Funny mode, even though the default value of `major-mode` is `nil`. By default, the value of `nil` for `major-mode` means to use the current buffer's major mode when creating new buffers (see Section 23.2.2 [Auto Major Mode], page 403), but with such `special` modes, Fundamental mode is used instead. Modes such as Dired, Rmail, and Buffer List use this feature.

  The function `view-buffer` does not enable View mode in buffers whose mode-class is special, because such modes usually provide their own View-like bindings.

  The `define-derived-mode` macro automatically marks the derived mode as special if the parent mode is special. Special mode is a convenient parent for such modes to inherit from; See Section 23.2.5 [Basic Major Modes], page 407.
- If you want to make the new mode the default for files with certain recognizable names, add an element to `auto-mode-alist` to select the mode for those file names (see Section 23.2.2 [Auto Major Mode], page 403). 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 215). If you do not autoload

the mode command, it is sufficient to add the element in the file that contains the mode definition.

- The top-level forms in the file defining the mode should be written so that they may be evaluated more than once without adverse consequences. For instance, use `defvar` or `defcustom` to set mode-related variables, so that they are not reinitialized if they already have a value (see Section 11.5 [Defining Variables], page 141).

23.2.2 How Emacs Chooses a Major Mode

When Emacs visits a file, it automatically selects a major mode for the buffer based on information in the file name or in the file itself. It also processes local variables specified in the file text.

`normal-mode` **&optional** *find-file* [Command]

This function establishes the proper major mode and buffer-local variable bindings for the current buffer. 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 11.11 [File Local Variables], page 156).

If the *find-file* argument to `normal-mode` is non-`nil`, `normal-mode` assumes that the `find-file` function is calling it. In this case, it may process local variables in the '`-*-`' line or at the end of the file. The variable `enable-local-variables` controls whether to do so. See Section "Local Variables in Files" in *The GNU Emacs Manual*, for the syntax of the local variables section of a file.

If you run `normal-mode` interactively, the argument *find-file* is normally `nil`. In this case, `normal-mode` unconditionally processes any file local variables.

The function calls `set-auto-mode` to choose a major mode. If this does not specify a mode, the buffer stays in the major mode determined by the default value of `major-mode` (see below).

`normal-mode` uses `condition-case` around the call to the major mode command, so errors are caught and reported as a '`File mode specification error`', followed by the original error message.

`set-auto-mode` **&optional** *keep-mode-if-same* [Function]

This function selects the major mode that is appropriate for the current buffer. It bases its decision (in order of precedence) on the '`-*-`' line, on any '`mode:`' local variable near the end of a file, on the '`#!`' line (using `interpreter-mode-alist`), on the text at the beginning of the buffer (using `magic-mode-alist`), and finally on the visited file name (using `auto-mode-alist`). See Section "How Major Modes are Chosen" in *The GNU Emacs Manual*. If `enable-local-variables` is `nil`, `set-auto-mode` does not check the '`-*-`' line, or near the end of the file, for any mode tag.

There are some file types where it is not appropriate to scan the file contents for a mode specifier. For example, a tar archive may happen to contain, near the end of the file, a member file that has a local variables section specifying a mode for that particular file. This should not be applied to the containing tar file. Similarly, a tiff image file might just happen to contain a first line that seems to match the '`-*-`'

pattern. For these reasons, both these file extensions are members of the list *inhibit-local-variables-regexps*. Add patterns to this list to prevent Emacs searching them for local variables of any kind (not just mode specifiers).

If *keep-mode-if-same* is non-`nil`, this function does not call the mode command if the buffer is already in the proper major mode. For instance, `set-visited-file-name` sets this to `t` to avoid killing buffer local variables that the user may have set.

`set-buffer-major-mode` *buffer* [Function]

This function sets the major mode of *buffer* to the default value of `major-mode`; if that is `nil`, it uses the current buffer's major mode (if that is suitable). As an exception, if *buffer*'s name is '`*scratch*`', it sets the mode to `initial-major-mode`.

The low-level primitives for creating buffers do not use this function, but medium-level commands such as `switch-to-buffer` and `find-file-noselect` use it whenever they create buffers.

`initial-major-mode` [User Option]

The value of this variable determines the major mode of the initial '`*scratch*`' buffer. The value should be a symbol that is a major mode command. The default value is `lisp-interaction-mode`.

`interpreter-mode-alist` [Variable]

This variable specifies major modes to use for scripts that specify a command interpreter in a '`#!`' line. Its value is an alist with elements of the form `(`*`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 486), with version numbers and backup suffixes removed using `file-name-sans-versions` (see Section 25.8.1 [File Name Components], page 482), 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))
```

23.2.3 Getting Help about a Major Mode

The `describe-mode` function provides information about major modes. It is normally bound to *C-h m*. It uses the value of the variable `major-mode` (see Section 23.2 [Major Modes], page 399), which is why every major mode command needs to set that variable.

`describe-mode` **&optional** *buffer* [Command]

This command displays the documentation of the current buffer's major mode and minor modes. It uses the `documentation` function to retrieve the documentation strings of the major and minor mode commands (see Section 24.2 [Accessing Documentation], page 452).

If called from Lisp with a non-nil *buffer* argument, this function displays the documentation for that buffer's major and minor modes, rather than those of the current buffer.

23.2.4 Defining Derived Modes

The recommended way to define a new major mode is to derive it from an existing one using `define-derived-mode`. If there is no closely related mode, you should inherit from either `text-mode`, `special-mode`, or `prog-mode`. See Section 23.2.5 [Basic Major Modes], page 407. If none of these are suitable, you can inherit from `fundamental-mode` (see Section 23.2 [Major Modes], page 399).

`define-derived-mode` *variant parent name docstring keyword-args. . . body. . .* [Macro]

This macro defines *variant* as a major mode command, using *name* as the string form of the mode name. *variant* and *parent* should be unquoted symbols.

The new command *variant* is defined to call the function *parent*, then override certain aspects of that parent mode:

- The new mode has its own sparse keymap, named `*variant*-map`. `define-derived-mode` makes the parent mode's keymap the parent of the new map, unless `*variant*-map` is already set and already has a parent.
- The new mode has its own syntax table, kept in the variable `*variant*-syntax-table`, unless you override this using the `:syntax-table` keyword (see below). `define-derived-mode` makes the parent mode's syntax-table the parent of `*variant*-syntax-table`, unless the latter is already set and already has a parent different from the standard syntax table.
- The new mode has its own abbrev table, kept in the variable `*variant*-abbrev-table`, unless you override this using the `:abbrev-table` keyword (see below).
- The new mode has its own mode hook, `*variant*-hook`. It runs this hook, after running the hooks of its ancestor modes, with `run-mode-hooks`, as the last thing it does. See Section 23.2.6 [Mode Hooks], page 408.

In addition, you can specify how to override other aspects of *parent* with *body*. The command *variant* evaluates the forms in *body* after setting up all its usual overrides, just before running the mode hooks.

If *parent* has a non-`nil` `mode-class` symbol property, then `define-derived-mode` sets the `mode-class` property of *variant* to the same value. This ensures, for example, that if *parent* is a special mode, then *variant* is also a special mode (see Section 23.2.1 [Major Mode Conventions], page 399).

You can also specify `nil` for *parent*. This gives the new mode no parent. Then `define-derived-mode` behaves as described above, but, of course, omits all actions connected with *parent*.

The argument *docstring* specifies the documentation string for the new mode. `define-derived-mode` adds some general information about the mode's hook, followed by the mode's keymap, at the end of this documentation string. If you omit *docstring*, `define-derived-mode` generates a documentation string.

The *keyword-args* are pairs of keywords and values. The values 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.

`derived-mode-p` &rest *modes* [Function]

This function returns non-`nil` if the current major mode is derived from any of the major modes given by the symbols *modes*.

23.2.5 Basic Major Modes

Apart from Fundamental mode, there are three major modes that other major modes commonly derive from: Text mode, Prog mode, and Special mode. While Text mode is useful in its own right (e.g. for editing files ending in '`.txt`'), Prog mode and Special mode exist mainly to let other modes derive from them.

As far as possible, new major modes should be derived, either directly or indirectly, from one of these three modes. One reason is that this allows users to customize a single mode hook (e.g. `prog-mode-hook`) for an entire family of relevant modes (e.g. all programming language modes).

`text-mode` [Command]

Text mode is a major mode for editing human languages. It defines the '`"`' and '`\`' characters as having punctuation syntax (see Section 35.2.1 [Syntax Class Table], page 235, vol. 2), and binds *`M-TAB`* to `ispell-complete-word` (see Section "Spelling" in *The GNU Emacs Manual*).

An example of a major mode derived from Text mode is HTML mode. See Section "SGML and HTML Modes" in *The GNU Emacs Manual*.

`prog-mode` [Command]

Prog mode is a basic major mode for buffers containing programming language source code. Most of the programming language major modes built into Emacs are derived from it.

Prog mode binds `parse-sexp-ignore-comments` to `t` (see Section 35.6.1 [Motion via Parsing], page 242, vol. 2) and `bidi-paragraph-direction` to `left-to-right` (see Section 38.23 [Bidirectional Display], page 382, vol. 2).

`special-mode` [Command]

Special mode is a basic major mode for buffers containing text that is produced specially by Emacs, rather than directly from a file. Major modes derived from Special mode are given a `mode-class` property of `special` (see Section 23.2.1 [Major Mode Conventions], page 399).

Special mode sets the buffer to read-only. Its keymap defines several common bindings, including *q* for `quit-window`, *z* for `kill-this-buffer`, and *g* for `revert-buffer` (see Section 26.3 [Reverting], page 510).

An example of a major mode derived from Special mode is Buffer Menu mode, which is used by the '`*Buffer List*`' buffer. See Section "Listing Existing Buffers" in *The GNU Emacs Manual*.

In addition, modes for buffers of tabulated data can inherit from Tabulated List mode, which is in turn derived from Special mode. See Section 23.2.7 [Tabulated List Mode], page 409.

23.2.6 Mode Hooks

Every major mode command should finish by running the mode-independent normal hook `change-major-mode-after-body-hook`, its mode hook, and the normal hook `after-change-major-mode-hook`. It does this by calling `run-mode-hooks`. If the major mode is a derived mode, that is if it calls another major mode (the parent mode) in its body, it should do this inside `delay-mode-hooks` so that the parent won't run these hooks itself. Instead, the derived mode's call to `run-mode-hooks` runs the parent's mode hook too. See Section 23.2.1 [Major Mode Conventions], page 399.

Emacs versions before Emacs 22 did not have `delay-mode-hooks`. Versions before 24 did not have `change-major-mode-after-body-hook`. When user-implemented major modes do not use `run-mode-hooks` and have not been updated to use these newer features, they won't entirely follow these conventions: they may run the parent's mode hook too early, or fail to run `after-change-major-mode-hook`. If you encounter such a major mode, please correct it to follow these conventions.

When you 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 396), but it also runs `change-major-mode-after-body-hook` and `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.

`change-major-mode-after-body-hook` [Variable]

This is a normal hook run by `run-mode-hooks`. It is run before the mode hooks.

`after-change-major-mode-hook` [Variable]

This is a normal hook run by `run-mode-hooks`. It is run at the very end of every properly-written major mode command.

23.2.7 Tabulated List mode

Tabulated List mode is a major mode for displaying tabulated data, i.e. data consisting of *entries*, each entry occupying one row of text with its contents divided into columns. Tabulated List mode provides facilities for pretty-printing rows and columns, and sorting the rows according to the values in each column. It is derived from Special mode (see Section 23.2.5 [Basic Major Modes], page 407).

Tabulated List mode is intended to be used as a parent mode by a more specialized major mode. Examples include Process Menu mode (see Section 37.6 [Process Information], page 266, vol. 2) and Package Menu mode (see Section "Package Menu" in *The GNU Emacs Manual*).

Such a derived mode should use `define-derived-mode` in the usual way, specifying `tabulated-list-mode` as the second argument (see Section 23.2.4 [Derived Modes], page 405). The body of the `define-derived-mode` form should specify the format of the tabulated data, by assigning values to the variables documented below; then, it should call the function `tabulated-list-init-header` to initialize the header line.

The derived mode should also define a *listing command*. This, not the mode command, is what the user calls (e.g. *`M-x list-processes`*). The listing command should create or switch to a buffer, turn on the derived mode, specify the tabulated data, and finally call `tabulated-list-print` to populate the buffer.

`tabulated-list-format` [Variable]

This buffer-local variable specifies the format of the Tabulated List data. Its value should be a vector. Each element of the vector represents a data column, and should be a list `(`*`name width sort`*`)`, where

- *name* is the column's name (a string).
- *width* is the width to reserve for the column (an integer). This is meaningless for the last column, which runs to the end of each line.
- *sort* specifies how to sort entries by the column. If `nil`, the column cannot be used for sorting. If `t`, the column is sorted by comparing string values. Otherwise, this should be a predicate function for `sort` (see Section 5.6.3 [Rearrangement], page 76), which accepts two arguments with the same form as the elements of `tabulated-list-entries` (see below).

`tabulated-list-entries` [Variable]

This buffer-local variable specifies the entries displayed in the Tabulated List buffer. Its value should be either a list, or a function.

If the value is a list, each list element corresponds to one entry, and should have the form `(`*id contents*`)`, where

- *id* is either `nil`, or a Lisp object that identifies the entry. If the latter, the cursor stays on the "same" entry when re-sorting entries. Comparison is done with `equal`.
- *contents* is a vector with the same number of elements as `tabulated-list-format`. Each vector element is either a string, which is inserted into the buffer as-is, or a list `(`*label* `.` *properties*`)`, which means to insert a text button by calling `insert-text-button` with *label* and *properties* as arguments (see Section 38.17.3 [Making Buttons], page 368, vol. 2).

  There should be no newlines in any of these strings.

Otherwise, the value should be a function which returns a list of the above form when called with no arguments.

`tabulated-list-revert-hook` [Variable]

This normal hook is run prior to reverting a Tabulated List buffer. A derived mode can add a function to this hook to recompute `tabulated-list-entries`.

`tabulated-list-printer` [Variable]

The value of this variable is the function called to insert an entry at point, including its terminating newline. The function should accept two arguments, *id* and *contents*, having the same meanings as in `tabulated-list-entries`. The default value is a function which inserts an entry in a straightforward way; a mode which uses Tabulated List mode in a more complex way can specify another function.

`tabulated-list-sort-key` [Variable]

The value of this variable specifies the current sort key for the Tabulated List buffer. If it is `nil`, no sorting is done. Otherwise, it should have the form `(`*name* `.` *flip*`)`, where *name* is a string matching one of the column names in `tabulated-list-format`, and *flip*, if non-`nil`, means to invert the sort order.

`tabulated-list-init-header` [Function]

This function computes and sets `header-line-format` for the Tabulated List buffer (see Section 23.4.7 [Header Lines], page 426), and assigns a keymap to the header line to allow sort entries by clicking on column headers.

Modes derived from Tabulated List mode should call this after setting the above variables (in particular, only after setting `tabulated-list-format`).

`tabulated-list-print` **&optional** *remember-pos* [Function]

This function populates the current buffer with entries. It should be called by the listing command. It erases the buffer, sorts the entries specified by `tabulated-list-entries` according to `tabulated-list-sort-key`, then calls the function specified by `tabulated-list-printer` to insert each entry.

If the optional argument *remember-pos* is non-`nil`, this function looks for the *id* element on the current line, if any, and tries to move to that entry after all the entries are (re)inserted.

23.2.8 Generic Modes

Generic modes are simple major modes with basic support for comment syntax and Font Lock mode. To define a generic mode, use the macro `define-generic-mode`. See the file '`generic-x.el`' for some examples of the use of `define-generic-mode`.

`define-generic-mode` *mode comment-list keyword-list font-lock-list auto-mode-list function-list* **&optional** *docstring* [Macro]

This macro defines a generic mode command named *mode* (a symbol, not quoted). The optional argument *docstring* is the documentation for the mode command. If you do not supply it, `define-generic-mode` generates one by default.

The argument *comment-list* is a list in which each element is either a character, a string of one or two characters, or a cons cell. A character or a string is set up in the mode's syntax table as a "comment starter". If the entry is a cons cell, the CAR is set up as a "comment starter" and the CDR as a "comment ender". (Use `nil` for the latter if you want comments to end at the end of the line.) Note that the syntax table mechanism has limitations about what comment starters and enders are actually possible. See Chapter 35 [Syntax Tables], page 234, vol. 2.

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 430.

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`.

23.2.9 Major Mode Examples

Text mode is perhaps the simplest mode besides Fundamental mode. Here are excerpts from '`text-mode.el`' that illustrate many of the conventions listed above:

```
;; Create the syntax table for this mode.
(defvar text-mode-syntax-table
  (let ((st (make-syntax-table)))
    (modify-syntax-entry ?\" ".   " st)
    (modify-syntax-entry ?\\ ".   " st)
    ;; Add 'p' so M-c on 'hello' leads to 'Hello', not 'hello'.
    (modify-syntax-entry ?' "w p" st)
    st)
  "Syntax table used while in 'text-mode'.")

;; Create the keymap for this mode.
```

```
(defvar text-mode-map
  (let ((map (make-sparse-keymap)))
    (define-key map "\e\t" 'ispell-complete-word)
    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'."
  (set (make-local-variable 'text-mode-variant) t)
  (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.)

The three Lisp modes (Lisp mode, Emacs Lisp mode, and Lisp Interaction mode) have more features than Text mode and the code is correspondingly more complicated. Here are excerpts from '`lisp-mode.el`' that illustrate how these modes are written.

Here is how the Lisp mode syntax and abbrev tables are defined:

```
;; Create mode-specific table variables.
(defvar lisp-mode-abbrev-table nil)
(define-abbrev-table 'lisp-mode-abbrev-table ())

(defvar lisp-mode-syntax-table
  (let ((table (copy-syntax-table emacs-lisp-mode-syntax-table)))
    (modify-syntax-entry ?\[ "_   " table)
    (modify-syntax-entry ?\] "_   " table)
    (modify-syntax-entry ?# "' 14" table)
    (modify-syntax-entry ?| "\" 23bn" table)
    table)
  "Syntax table used in `lisp-mode'.")
```

The three modes for Lisp share much of their code. For instance, each calls the following function to set various variables:

```
(defun lisp-mode-variables (&optional syntax keywords-case-insensitive)
  (when syntax
    (set-syntax-table lisp-mode-syntax-table))
  (setq local-abbrev-table lisp-mode-abbrev-table)
  ...
```

Amongst other things, this function sets up the `comment-start` variable to handle Lisp comments:

```
  (make-local-variable 'comment-start)
  (setq comment-start ";")
  ...
```

Each of the different Lisp modes has a slightly different keymap. For example, Lisp mode binds *C-c C-z* to `run-lisp`, but the other Lisp modes do not. However, all Lisp modes have some commands in common. The following code sets up the common commands:

```
(defvar lisp-mode-shared-map
  (let ((map (make-sparse-keymap)))
    (define-key map "\e\C-q" 'indent-sexp)
    (define-key map "\177" 'backward-delete-char-untabify)
    map)
  "Keymap for commands shared by all sorts of Lisp modes.")
```

And here is the code to set up the keymap for Lisp mode:

```
(defvar lisp-mode-map
  (let ((map (make-sparse-keymap))
(menu-map (make-sparse-keymap "Lisp")))
    (set-keymap-parent map lisp-mode-shared-map)
    (define-key map "\e\C-x" 'lisp-eval-defun)
    (define-key map "\C-c\C-z" 'run-lisp)
    ...
    map)
  "Keymap for ordinary Lisp mode.
All commands in 'lisp-mode-shared-map' are inherited by this map.")
```

Finally, here is the major mode command for Lisp mode:

```
(define-derived-mode lisp-mode prog-mode "Lisp"
  "Major mode for editing Lisp code for Lisps other than GNU Emacs Lisp.
Commands:
Delete converts tabs to spaces as it moves back.
Blank lines separate paragraphs.  Semicolons start comments.

\\{lisp-mode-map}
Note that 'run-lisp' may be used either to start an inferior Lisp job
or to switch back to an existing one.

Entry to this mode calls the value of 'lisp-mode-hook'
if that value is non-nil."
  (lisp-mode-variables nil t)
  (set (make-local-variable 'find-tag-default-function)
       'lisp-find-tag-default)
  (set (make-local-variable 'comment-start-skip)
       "\\(\\(^\\|[^\\\\\n]\\)\\(\\\\\\\\\\)*\\)\\(;+\\|#|\\) *")
  (setq imenu-case-fold-search t))
```

23.3 Minor Modes

A *minor mode* provides optional features that users may enable or disable independently of the choice of major mode. Minor modes can be enabled individually or in combination.

Most minor modes implement features that are independent of the major mode, and can thus be used with most major modes. For example, Auto Fill mode works with any major mode that permits text insertion. A few minor modes, however, are specific to a particular major mode. For example, Diff Auto Refine mode is a minor mode that is intended to be used only with Diff mode.

Ideally, a minor mode should have its desired effect regardless of the other minor modes in effect. It should be possible to activate and deactivate minor modes in any order.

`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. These conventions are described below. The easiest way to follow them is to use the macro `define-minor-mode`. See Section 23.3.3 [Defining Minor Modes], page 416.

- Define a variable whose name ends in '`-mode`'. We call this the *mode variable*. The minor mode command should set this variable. The value will be `nil` is the mode is disabled, and non-`nil` if the mode is enabled. The variable should be buffer-local if the minor mode is buffer-local.

  This variable is used in conjunction with the `minor-mode-alist` to display the minor mode name in the mode line. It also determines whether the minor mode keymap is active, via `minor-mode-map-alist` (see Section 22.9 [Controlling Active Maps], page 369). Individual commands or hooks can also check its value.

- Define a command, called the *mode command*, whose name is the same as the mode variable. Its job is to set the value of the mode variable, plus anything else that needs to be done to actually enable or disable the mode's features.

  The mode command should accept one optional argument. If called interactively with no prefix argument, it should toggle the mode (i.e. enable if it is disabled, and disable if it is enabled). If called interactively with a prefix argument, it should enable the mode if the argument is positive and disable it otherwise.

  If the mode command is called from Lisp (i.e. non-interactively), it should enable the mode if the argument is omitted or `nil`; it should toggle the mode if the argument is the symbol `toggle`; otherwise it should treat the argument in the same way as for an interactive call with a numeric prefix argument, as described above.

  The following example shows how to implement this behavior (it is similar to the code generated by the `define-minor-mode` macro):

  ```
  (interactive (list (or current-prefix-arg 'toggle)))
  (let ((enable (if (eq arg 'toggle)
                    (not foo-mode) ; this mode's mode variable
                  (> (prefix-numeric-value arg) 0))))
    (if enable
        do-enable
      do-disable))
  ```

  The reason for this somewhat complex behavior is that it lets users easily toggle the minor mode interactively, and also lets the minor mode be easily enabled in a mode hook, like this:

  ```
  (add-hook 'text-mode-hook 'foo-mode)
  ```

  This behaves correctly whether or not `foo-mode` was already enabled, since the `foo-mode` mode command unconditionally enables the minor mode when it is called from Lisp with no argument. Disabling a minor mode in a mode hook is a little uglier:

  ```
  (add-hook 'text-mode-hook (lambda () (foo-mode -1)))
  ```

  However, this is not very commonly done.

- Add an element to `minor-mode-alist` for each minor mode (see [Definition of minor-mode-alist], page 423), if you want to indicate the minor mode in the mode line. This element should be a list of the following form:

```
(mode-variable string)
```

Here *mode-variable* is the variable that controls enabling of the minor mode, and *string* is a short string, starting with a space, to represent the mode in the mode line. These strings must be short so that there is room for several of them at once.

When you add an element to `minor-mode-alist`, use `assq` to check for an existing element, to avoid duplication. For example:

```
(unless (assq 'leif-mode minor-mode-alist)
  (push '(leif-mode " Leif") minor-mode-alist))
```

or like this, using `add-to-list` (see Section 5.5 [List Variables], page 71):

```
(add-to-list 'minor-mode-alist '(leif-mode " Leif"))
```

In addition, several major mode conventions apply to minor modes as well: those regarding the names of global symbols, the use of a hook at the end of the initialization function, and the use of keymaps and other tables.

The minor mode should, if possible, support enabling and disabling via Custom (see Chapter 14 [Customization], page 190). To do this, the mode variable should be defined with `defcustom`, usually with `:type 'boolean`. If just setting the variable is not sufficient to enable the mode, you should also specify a `:set` method which enables the mode by invoking the mode command. Note in the variable's documentation string that setting the variable other than via Custom may not take effect. Also, mark the definition with an autoload cookie (see [autoload cookie], page 215), and specify a `:require` so that customizing the variable will load the library that defines the mode. For example:

```
;;;###autoload
(defcustom msb-mode nil
  "Toggle msb-mode.
Setting this variable directly does not take effect;
use either \\[customize] or the function `msb-mode'."
  :set 'custom-set-minor-mode
  :initialize 'custom-initialize-default
  :version "20.4"
  :type    'boolean
  :group   'msb
  :require 'msb)
```

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 370.

One use of minor mode keymaps is to modify the behavior of certain self-inserting characters so that they do something else as well as self-insert. (Another way to customize `self-insert-command` is through `post-self-insert-hook`. Apart from this, the facilities for customizing `self-insert-command` are limited to special cases, designed for abbrevs and Auto Fill mode. Do not try substituting your own definition of `self-insert-command` for the standard one. The editor command loop handles this function specially.)

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.

`define-minor-mode` *mode doc* [*init-value* [*lighter* [*keymap*]]] *keyword-args... body...* [Macro]

This macro defines a new minor mode whose name is *mode* (a symbol). It defines a command named *mode* to toggle the minor mode, with *doc* as its documentation string.

The toggle command takes one optional (prefix) argument. If called interactively with no argument it toggles the mode on or off. A positive prefix argument enables the mode, any other prefix argument disables it. From Lisp, an argument of `toggle` toggles the mode, whereas an omitted or `nil` argument enables the mode. This makes it easy to enable the minor mode in a major mode hook, for example. If *doc* is nil, the macro supplies a default documentation string explaining the above.

By default, it also defines a variable named *mode*, which is set to `t` or `nil` by enabling or disabling the mode. The 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. If non-`nil`, it should be a variable name (whose value is a keymap), a keymap, or an alist of the form

```
(key-sequence . definition)
```

where each *key-sequence* and *definition* are arguments suitable for passing to `define-key` (see Section 22.12 [Changing Key Bindings], page 375). If *keymap* is a keymap or an alist, this also defines the variable *`mode`*`-map`.

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 192.

`:global` *`global`*
: If non-`nil`, this specifies that the minor mode should be global rather than buffer-local. It defaults to `nil`.

One of the effects of making a minor mode global is that the *mode* variable becomes a customization variable. Toggling it through the Customize interface turns the mode on and off, and its value can be saved for future Emacs sessions (see Section "Saving Customizations" in *The GNU Emacs Manual*. For the saved variable to work, you should ensure that the `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.

`:variable` *`place`*
: This replaces the default variable *mode*, used to store the state of the mode. If you specify this, the *mode* variable is not defined, and any *init-value* argument is unused. *place* can be a different named variable (which you must define yourself), or anything that can be used with the `setf` function (see Section "Generalized Variables" in *Common Lisp Extensions*). *place* can also be a cons `(`*`get`*` . `*`set`*`)`, where *get* is an expression that returns the current state, and *set* is a function of one argument (a state) that sets it.

`:after-hook` *`after-hook`*
: This defines a single Lisp form which is evaluated after the mode hooks have run. It should not be quoted.

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 then runs the mode hook variable *`mode`*`-hook` and finishes by evaluating any form in `:after-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.
Interactively with no argument, this command toggles the mode.
A positive prefix argument enables the mode, any other prefix
argument disables it.  From Lisp, argument omitted or nil enables
the mode, ‘toggle’ toggles the state.

When Hungry mode is enabled, the control delete key
gobbles all preceding whitespace except the last.
See the command \\[hungry-electric-delete]."
 ;; The initial value.
 nil
 ;; The indicator for the mode line.
```

```
       " Hungry"
       ;; The minor mode bindings.
       '(([C-backspace] . hungry-electric-delete))
       :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.
     ...rest of documentation as before..."
      ;; The initial value.
      :init-value nil
      ;; The indicator for the mode line.
      :lighter " Hungry"
      ;; The minor mode bindings.
      :keymap
      '(([C-backspace] . hungry-electric-delete)
        ([C-M-backspace]
         . (lambda ()
             (interactive)
             (hungry-electric-delete t))))
      :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 190), which can be toggled in the Customize interface to turn the minor mode on and off. As with `define-minor-mode`, you should ensure that the `define-globalized-minor-mode` form is evaluated each time Emacs starts, for example by providing a `:require` keyword.

Use `:group` *group* in *keyword-args* to specify the custom group for the mode variable of the global minor mode.

Generally speaking, when you define a globalized minor mode, you should also define a non-globalized version, so that people can use (or disable) it in individual buffers. This also allows them to disable a globally enabled minor mode in a specific major mode, by using that mode's hook.

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

The contents of each mode line are specified by the buffer-local variable `mode-line-format` (see Section 23.4.3 [Mode Line Top], page 421). This variable holds a *mode line construct*: a template that controls what is displayed on the buffer's mode line. The value of `header-line-format` specifies the buffer's header line in the same way. All windows for the same buffer use the same `mode-line-format` and `header-line-format`.

For efficiency, Emacs does not continuously recompute each window's mode line and header line. It does so when circumstances appear to call for it—for instance, if you change the window configuration, switch buffers, narrow or widen the buffer, scroll, or modify the buffer. If you alter any of the variables referenced by `mode-line-format` or `header-line-format` (see Section 23.4.4 [Mode Line Variables], page 422), or any other data structures that affect how text is displayed (see Chapter 38 [Display], page 299, vol. 2), you should use the function `force-mode-line-update` to update the display.

`force-mode-line-update` **&optional** *all* [Function]

> This function forces Emacs to update the current buffer's mode line and header line, based on the latest values of all relevant variables, during its next redisplay cycle. If the optional argument *all* is non-`nil`, it forces an update for all mode lines and header lines.
>
> This function also forces an update of the menu bar and frame title.

The selected window's mode line is usually displayed in a different color using the face `mode-line`. Other windows' mode lines appear in the face `mode-line-inactive` instead. See Section 38.12 [Faces], page 325, vol. 2.

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 29.5 [Frame Titles], page 81, vol. 2) and header lines (see Section 23.4.7 [Header Lines], page 426).

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 424.

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 425.

`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*.

`(`*`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` [User Option]

The value of this variable is a mode line construct that controls the contents of the mode-line. It is always buffer-local in all buffers.

If you set this variable to `nil` in a buffer, that buffer does not have a mode line. (A window that is just one line tall also does not display a mode line.)

The default value of `mode-line-format` is designed to use the values of other variables such as `mode-line-position` and `mode-line-modes` (which in turn incorporates the values of the variables `mode-name` and `minor-mode-alist`). Very few modes need to alter `mode-line-format` itself. For most purposes, it is sufficient to alter some of the variables that `mode-line-format` either directly or indirectly refers to.

If you do alter `mode-line-format` itself, the new value should use the same variables that appear in the default value (see Section 23.4.4 [Mode Line Variables], page 422), rather than duplicating their contents or displaying the information in another fashion. This way, customizations made by the user or by Lisp programs (such as `display-time` and major modes) via changes to those variables remain effective.

Here is a hypothetical example of a `mode-line-format` that might be useful for Shell mode (in reality, Shell mode does not set `mode-line-format`):

```
(setq mode-line-format
  (list "-"
   'mode-line-mule-info
   'mode-line-modified
   'mode-line-frame-identification
   "%b--"
   ;; Note that this is evaluated while making the list.
   ;; It makes a mode line construct which is just a string.
   (getenv "HOST")
   ":"
   'default-directory
   "   "
   'global-mode-string
   "   %[("
   '(:eval (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 the value of `mode-line-format` is changed to use them. However, various parts of Emacs set these variables on the understanding that they will control parts of the mode line; therefore, practically speaking, it is essential for the mode line to use them.

`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 182, vol. 2.

`mode-line-modified` [Variable]

This variable holds the value of the mode line construct that displays whether the current buffer is modified. Its default value displays '`**`' if the buffer is modified, '`--`' if the buffer is not modified, '`%%`' if the buffer is read only, and '`%*`' if the buffer is read only and modified.

Changing this variable does not force an update of the mode line.

`mode-line-frame-identification` [Variable]

This variable identifies the current frame. Its default value displays `" "` if you are using a window system which can show multiple frames, or `"-%F "` on an ordinary terminal which shows only one frame at a time.

`mode-line-buffer-identification` [Variable]

This variable identifies the buffer being displayed in the window. Its default value displays the buffer name, padded with spaces to at least 12 columns.

`mode-line-position` [User Option]

This variable indicates the position in the buffer. Its default value displays the buffer percentage and, optionally, the buffer size, the line number and the column number.

`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` [User Option]

This variable displays the buffer's major and minor modes. Its default value also displays the recursive editing level, information on the process status, and whether narrowing is in effect.

`mode-line-remote` [Variable]

This variable is used to show whether `default-directory` for the current buffer is remote.

`mode-line-client` [Variable]

This variable is used to identify `emacsclient` frames.

The following three variables are used in `mode-line-modes`:

`mode-name` [Variable]

This buffer-local variable holds the "pretty" name of the current buffer's major mode. Each major mode should set this variable so that the mode name will appear in the mode line. The value does not have to be a string, but can use any of the data types valid in a mode-line construct (see Section 23.4.2 [Mode Line Data], page 419). To compute the string that will identify the mode name in the mode line, use `format-mode-line` (see Section 23.4.8 [Emulating Mode Line], page 426).

`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:

```
(minor-mode-variable mode-line-string)
```

More generally, *mode-line-string* can be any mode line construct. It appears in the mode line when the value of *minor-mode-variable* is non-`nil`, and not otherwise. These strings should begin with spaces so that they don't run together. Conventionally, the *minor-mode-variable* for a specific mode is set to a non-`nil` value when that minor mode is activated.

`minor-mode-alist` itself is not buffer-local. Each variable mentioned in the alist should be buffer-local if its minor mode can be enabled separately in each buffer.

`global-mode-string` [Variable]

This variable holds a mode line construct that, by default, appears in the mode line 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`.

Here is a simplified version of the default value of `mode-line-format`. The real default value also specifies addition of text properties.

```
("-"
 mode-line-mule-info
 mode-line-modified
 mode-line-frame-identification
 mode-line-buffer-identification
 "   "
 mode-line-position
 (vc-mode vc-mode)
 "   "
 mode-line-modes
 (which-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 `buffer-name` function. See Section 27.3 [Buffer Names], page 4, vol. 2.

`%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 `buffer-file-name` function. See Section 27.4 [Buffer File Name], page 5, vol. 2.

`%F` The title (only on a window system) or the name of the selected frame. See Section 29.3.3.1 [Basic Parameters], page 71, vol. 2.

`%i` The size of the accessible part of the current buffer; basically `(- (point-max) (point-min))`.

`%I` Like '`%i`', but the size is printed in a more readable way by using '`k`' for 10^3, '`M`' for 10^6, '`G`' for 10^9, etc., to abbreviate.

`%l` The current line number of point, counting within the accessible portion of the buffer.

`%n` '`Narrow`' when narrowing is in effect; nothing otherwise (see `narrow-to-region` in Section 30.4 [Narrowing], page 109, vol. 2).

`%p` The percentage of the buffer text above the **top** of window, or '`Top`', '`Bottom`' or '`All`'. Note that the default mode line construct truncates this to three characters.

`%P` The percentage of the buffer text that is above the **bottom** of the window (which includes the text visible in the window, as well as the text above the top), plus '`Top`' if the top of the buffer is visible on screen; or '`Bottom`' or '`All`'.

`%s` The status of the subprocess belonging to the current buffer, obtained with `process-status`. See Section 37.6 [Process Information], page 266, vol. 2.

`%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 33.9.9 [MS-DOS File Types], page 205, vol. 2).

`%z` The mnemonics of keyboard, terminal, and buffer coding systems.

`%Z` Like '`%z`', but including the end-of-line format.

`%*` '`%`' if the buffer is read only (see `buffer-read-only`);
'`*`' if the buffer is modified (see `buffer-modified-p`);
'`-`' otherwise. See Section 27.5 [Buffer Modification], page 7, vol. 2.

`%+` '`*`' if the buffer is modified (see `buffer-modified-p`);
'`%`' if the buffer is read only (see `buffer-read-only`);
'`-`' otherwise. This differs from '`%*`' only for a modified read-only buffer. See Section 27.5 [Buffer Modification], page 7, vol. 2.

`%&` '`*`' 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 21.13 [Recursive Editing], page 355.

`%]` 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 `header-line-format`:

`header-line-format` [Variable]

This variable, local in every buffer, specifies how to display the header line, for windows displaying the buffer. The format of the value is the same as for `mode-line-format` (see Section 23.4.2 [Mode Line Data], page 419). 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 construct.

`format-mode-line` *format* **&optional** *face window buffer* [Function]

This function formats a line of text according to *format* as if it were generating the mode line for *window*, but it also returns the text as a string. The argument *window* defaults to the selected window. If *buffer* is non-`nil`, all the information used is taken from *buffer*; by default, it comes from *window*'s buffer.

The value string normally has text properties that correspond to the faces, keymaps, etc., that the mode line would have. Any character for which no `face` property is specified by *format* gets a default value determined by *face*. If *face* is `t`, that stands for either `mode-line` if *window* is selected, otherwise `mode-line-inactive`. If *face* is `nil` or omitted, that stands for the default face. If *face* is an integer, the value returned by this function will have no text properties.

You can also specify other valid faces as the value of *face*. If specified, that face provides the `face` property for characters whose face is not specified by *format*.

Note that using `mode-line`, `mode-line-inactive`, or `header-line` as *face* will actually redisplay the mode line or the header line, respectively, using the current definitions of the corresponding face, in addition to returning the formatted string. (Other faces do not cause redisplay.)

For example, `(format-mode-line header-line-format)` returns the text that would appear in the selected window's header line (`""` if it has no header line). `(format-mode-line header-line-format 'header-line)` returns the same text, with each character carrying the face that it will have in the header line itself, and also redraws the header line.

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* [Command]

This function defines a local menu bar item named *name* to run Imenu.

The user-level commands for using Imenu are described in the Emacs Manual (see Section "Imenu" in *the Emacs Manual*). This section explains how to customize Imenu's method of finding definitions or buffer portions for a particular major mode.

The usual and simplest way is to set the variable `imenu-generic-expression`:

`imenu-generic-expression` [Variable]

This variable, if non-`nil`, is a list that specifies regular expressions for finding definitions for Imenu. Simple elements of `imenu-generic-expression` look like this:

```
(menu-title regexp index)
```

Here, if *menu-title* is non-`nil`, it says that the matches for this element should go in a submenu of the buffer index; *menu-title* itself specifies the name for the submenu. If *menu-title* is `nil`, the matches for this element go directly in the top level of the buffer index.

The second item in the list, *regexp*, is a regular expression (see Section 34.3 [Regular Expressions], page 211, vol. 2); anything in the buffer that it matches is considered a definition, something to mention in the buffer index. The third item, *index*, is a non-negative integer that indicates which subexpression in *regexp* matches the definition's name.

An element can also look like this:

```
(menu-title regexp index function arguments...)
```

Each match for this element creates an index item, and when the index item is selected by the user, it calls *function* with arguments consisting of the item name, the buffer position, and *arguments*.

For Emacs Lisp mode, `imenu-generic-expression` could look like this:

```
((nil "^\\s-*(def\\(un\\|subst\\|macro\\|advice\\)\
\\s-+\\([-A-Za-z0-9+]+\\)" 2)
 ("*Vars*" "^\\s-*(def\\(var\\|const\\)\
\\s-+\\([-A-Za-z0-9+]+\\)" 2)
```

```
 ("*Types*"
  "^\\s-*\
(def\\(type\\|struct\\|class\\|ine-condition\\)\
\\s-+\\([-A-Za-z0-9+]+\\)" 2))
```

Setting this variable makes it buffer-local in the current buffer.

imenu-case-fold-search [Variable]

This variable controls whether matching against the regular expressions in the value of `imenu-generic-expression` is case-sensitive: `t`, the default, means matching should ignore case.

Setting this variable makes it buffer-local in the current buffer.

imenu-syntax-alist [Variable]

This variable is an alist of syntax table modifiers to use while processing `imenu-generic-expression`, to override the syntax table of the current buffer. Each element should have this form:

```
(characters . syntax-description)
```

The CAR, *characters*, can be either a character or a string. The element says to give that character or characters the syntax specified by *syntax-description*, which is passed to `modify-syntax-entry` (see Section 35.3 [Syntax Table Functions], page 238, vol. 2).

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 buffer-local minor mode that automatically attaches `face` properties to certain parts of the buffer based on their syntactic role. How it parses the buffer depends on the major mode; most major modes define syntactic criteria for which faces to use in which contexts. This section explains how to customize Font Lock for a particular major mode.

Font Lock mode finds text to highlight in 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 to specify how to fontify text in that mode. It automatically becomes buffer-local when set. If its value is `nil`, Font Lock mode does no highlighting, and you can use the '`Faces`' menu (under '`Edit`' and then '`Text Properties`' in the menu bar) to assign faces explicitly to text in the buffer.

If non-`nil`, the value should look like this:

```
(keywords [keywords-only [case-fold
 [syntax-alist [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 the '`mode default`' level of fontification, the next symbol level 1 fontification, the next level 2, and so on. The '`mode default`' level is normally the same as level 1. It is used when `font-lock-maximum-decoration` has a `nil` value. See Section 23.6.5 [Levels of Font Lock], page 436.

The second element, *keywords-only*, specifies the value of the variable `font-lock-keywords-only`. If this is omitted or `nil`, syntactic fontification (of strings and comments) is also performed. If this is non-`nil`, syntactic fontification is not performed. See Section 23.6.8 [Syntactic Font Lock], page 437.

The third element, *case-fold*, specifies the value of `font-lock-keywords-case-fold-search`. If it is non-`nil`, Font Lock mode ignores case during search-based fontification.

If the fourth element, *syntax-alist*, is non-`nil`, it should be a list of cons cells of the form `(char-or-string . string)`. These are used to set up a syntax table for syntactic fontification; the resulting syntax table is stored in `font-lock-syntax-table`. If *syntax-alist* is omitted or `nil`, syntactic fontification uses the syntax table returned by the `syntax-table` function. See Section 35.3 [Syntax Table Functions], page 238, vol. 2.

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 435.

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 variable which directly controls search-based fontification is `font-lock-keywords`, which is typically specified via the *keywords* element in `font-lock-defaults`.

`font-lock-keywords` [Variable]

The value of this variable is a list of the keywords to highlight. Lisp programs should not set this variable directly. Normally, the value is automatically set by Font Lock mode, using the *keywords* element in `font-lock-defaults`. The value can also be altered using the functions `font-lock-add-keywords` and `font-lock-remove-keywords` (see Section 23.6.3 [Customizing Keywords], page 434).

Each element of `font-lock-keywords` specifies how to find certain cases of text, and how to highlight those cases. Font Lock mode processes the elements of `font-lock-keywords` one by one, and for each element, it finds and handles all matches. Ordinarily, once part of the text has been fontified already, this cannot be overridden by a subsequent match in the same text; but you can specify different behavior using the *override* element of a *subexp-highlighter*.

Each element of `font-lock-keywords` should have one of these forms:

`regexp`
Highlight all matches for *regexp* using `font-lock-keyword-face`. For example,

```
;; Highlight occurrences of the word 'foo'
;; using font-lock-keyword-face.
"\\<foo\\>"
```

Be careful when composing these regular expressions; a poorly written pattern can dramatically slow things down! The function `regexp-opt` (see Section 34.3.3 [Regexp Functions], page 220, vol. 2) is useful for calculating optimal regular expressions to match several keywords.

`function`
Find text by calling *function*, and highlight the matches it finds using `font-lock-keyword-face`.

When *function* is called, it receives one argument, the limit of the search; it should begin searching at point, and not search beyond the limit. It should return non-`nil` if it succeeds, and set the match data to describe the match that was found. Returning `nil` indicates failure of the search.

Fontification will call *function* repeatedly with the same limit, and with point where the previous invocation left it, until *function* fails. On failure, *function* need not reset point in any particular way.

`(`*`matcher`*` . `*`subexp`*`)`
In this kind of element, *matcher* is either a regular expression or a function, as described above. The CDR, *subexp*, specifies which subexpression of *matcher* should be highlighted (instead of the entire text that *matcher* matched).

```
;; Highlight the 'bar' in each occurrence of 'fubar',
;; using font-lock-keyword-face.
("fu\\(bar\\)" . 1)
```

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 220, vol. 2) 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 435.

`(`*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 23.6.9 [Multiline Font Lock], page 438.

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 commands must not call `font-lock-add-keywords` under any circumstances, either directly or indirectly, except through their mode hooks. (Doing so would lead to incorrect behavior for some minor modes.) They should set up their rules for search-based fontification by setting `font-lock-keywords`.

`font-lock-remove-keywords` *mode keywords* [Function]

This function removes *keywords* from `font-lock-keywords` for the current buffer or for major mode *mode*. As in `font-lock-add-keywords`, *mode* should be a major mode command name or `nil`. All the caveats and requirements for `font-lock-add-keywords` apply here too.

For example, the following code adds two fontification patterns for C mode: one to fontify the word '`FIXME`', even in comments, and another to fontify the words '`and`', '`or`' and '`not`' as keywords.

```
(font-lock-add-keywords 'c-mode
 '(("\\<\\(FIXME\\):" 1 font-lock-warning-face prepend)
   ("\\<\\(and\\|or\\|not\\)\\>" . font-lock-keyword-face)))
```

This example affects only C mode proper. To add the same patterns to C mode *and* all modes derived from it, do this instead:

```
(add-hook 'c-mode-hook
 (lambda ()
  (font-lock-add-keywords nil
   '(("\\<\\(FIXME\\):" 1 font-lock-warning-face prepend)
     ("\\<\\(and\\|or\\|not\\)\\>" .
      font-lock-keyword-face)))))
```

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 429).

`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 430.

`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`.

`jit-lock-register` *function* **&optional** *contextual* [Function]

This function tells Font Lock mode to run the Lisp function *function* any time it has to fontify or refontify part of the current buffer. It calls *function* before calling the default fontification functions, and gives it two arguments, *start* and *end*, which specify the region to be fontified or refontified.

The optional argument *contextual*, if non-`nil`, forces Font Lock mode to always refontify a syntactically relevant part of the buffer, and not just the modified lines. This argument can usually be omitted.

`jit-lock-unregister` *function* [Function]

If *function* was previously registered as a fontification function using `jit-lock-register`, this function unregisters it.

23.6.5 Levels of Font Lock

Some major modes offer three different levels of fontification. You can define multiple levels by using a list of symbols for *keywords* in `font-lock-defaults`. Each symbol specifies one level of fontification; it is up to the user to choose one of these levels, normally by setting `font-lock-maximum-decoration` (see Section "Font Lock" in *the GNU Emacs Manual*). The chosen level's symbol value is used to initialize `font-lock-keywords`.

Here are the conventions for how to define the levels of fontification:

- Level 1: highlight function declarations, file directives (such as include or import directives), strings and comments. The idea is speed, so only the most important and top-level components are fontified.
- Level 2: in addition to level 1, highlight all language keywords, including type names that act like keywords, as well as named constant values. The idea is that all keywords (either syntactic or semantic) should be fontified appropriately.
- Level 3: in addition to level 2, highlight the symbols being defined in function and variable declarations, and all builtin function names, wherever they appear.

23.6.6 Precalculated Fontification

Some major modes such as `list-buffers` and `occur` construct the buffer text programmatically. The easiest way for them to support Font Lock mode is to specify the faces of text when they insert the text in the buffer.

The way to do this is to specify the faces in the text with the special text property `font-lock-face` (see Section 32.19.4 [Special Properties], page 162, vol. 2). When Font Lock mode is enabled, this property controls the display, just like the `face` property. When Font Lock mode is disabled, `font-lock-face` has no effect on the display.

It is ok for a mode to use `font-lock-face` for some text and also use the normal Font Lock machinery. But if the mode does not use the normal Font Lock machinery, it should not set the variable `font-lock-defaults`.

23.6.7 Faces for Font Lock

Font Lock mode can highlight using any face, but Emacs defines several faces specifically for Font Lock to use to highlight text. These *Font Lock faces* are listed below. They can also be used by major modes for syntactic highlighting outside of Font Lock mode (see Section 23.2.1 [Major Mode Conventions], page 399).

Each of these symbols is both a face name, and a variable whose default value is the symbol itself. Thus, the default value of `font-lock-comment-face` is `font-lock-comment-face`.

The faces are listed with descriptions of their typical usage, and in order of greater to lesser “prominence”. If a mode’s syntactic categories do not fit well with the usage descriptions, the faces can be assigned using the ordering as a guide.

`font-lock-warning-face`
: for a construct that is peculiar, or that greatly changes the meaning of other text, like ‘`;;;###autoload`’ in Emacs Lisp and ‘`#error`’ in C.

`font-lock-function-name-face`
: for the name of a function being defined or declared.

`font-lock-variable-name-face`
: for the name of a variable being defined or declared.

`font-lock-keyword-face`
: for a keyword with special syntactic significance, like ‘`for`’ and ‘`if`’ in C.

`font-lock-comment-face`
: for comments.

`font-lock-comment-delimiter-face`
: for comments delimiters, like ‘`/*`’ and ‘`*/`’ in C. On most terminals, this inherits from `font-lock-comment-face`.

`font-lock-type-face`
: for the names of user-defined data types.

`font-lock-constant-face`
: for the names of constants, like ‘`NULL`’ in C.

`font-lock-builtin-face`
: for the names of built-in functions.

`font-lock-preprocessor-face`
: for preprocessor commands. This inherits, by default, from `font-lock-builtin-face`.

`font-lock-string-face`
: for string constants.

`font-lock-doc-face`
: for documentation strings in the code. This inherits, by default, from `font-lock-string-face`.

`font-lock-negation-char-face`
: for easily-overlooked negation characters.

23.6.8 Syntactic Font Lock

Syntactic fontification uses a syntax table (see Chapter 35 [Syntax Tables], page 234, vol. 2) to find and highlight syntactically relevant text. If enabled, it runs prior to search-based fontification. The variable `font-lock-syntactic-face-function`, documented below, determines which syntactic constructs to highlight. There are several variables that affect syntactic fontification; you should set them by means of `font-lock-defaults` (see Section 23.6.1 [Font Lock Basics], page 429).

Whenever Font Lock mode performs syntactic fontification on a stretch of text, it first calls the function specified by `syntax-propertize-function`. Major modes can use this to apply `syntax-table` text properties to override the buffer's syntax table in special cases. See Section 35.4 [Syntax Properties], page 240, vol. 2.

`font-lock-keywords-only` [Variable]

If the value of this variable is non-`nil`, Font Lock does not do syntactic fontification, only search-based fontification based on `font-lock-keywords`. It is normally set by Font Lock mode based on the *keywords-only* element in `font-lock-defaults`.

`font-lock-syntax-table` [Variable]

This variable holds the syntax table to use for fontification of comments and strings. It is normally set by Font Lock mode based on the *syntax-alist* element in `font-lock-defaults`. If this value is `nil`, syntactic fontification uses the buffer's syntax table (the value returned by the function `syntax-table`; see Section 35.3 [Syntax Table Functions], page 238, vol. 2).

`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. The value is normally set through an *other-vars* element in `font-lock-defaults`. If it is `nil`, Font Lock uses `syntax-begin-function` to move back outside of any comment, string, or sexp (see Section 35.6.2 [Position Parse], page 243, vol. 2).

This variable is semi-obsolete; we usually recommend setting `syntax-begin-function` instead. One of its uses is to tune the behavior of syntactic fontification, e.g. to ensure that different kinds of strings or comments are highlighted differently.

The specified 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.

`font-lock-syntactic-face-function` [Variable]

If this variable is non-`nil`, it should be a function to determine which face to use for a given syntactic element (a string or a comment). The value is normally set through an *other-vars* element in `font-lock-defaults`.

The function is called with one argument, the parse state at point returned by `parse-partial-sexp`, and should return a face. The default value returns `font-lock-comment-face` for comments and `font-lock-string-face` for strings (see Section 23.6.7 [Faces for Font Lock], page 436).

23.6.9 Multiline Font Lock Constructs

Normally, elements of `font-lock-keywords` should not match across multiple lines; that doesn't work reliably, because Font Lock usually scans just part of the buffer, and it can miss a multi-line construct that crosses the line boundary where the scan starts. (The scan normally starts at the beginning of a line.)

Making elements that match multiline constructs work properly has two aspects: correct *identification* and correct *rehighlighting*. The first means that Font Lock finds all multiline constructs. The second means that Font Lock will correctly rehighlight all the relevant text when a multiline construct is changed—for example, if some of the text that was previously part of a multiline construct ceases to be part of it. The two aspects are closely related, and often getting one of them to work will appear to make the other also work. However, for reliable results you must attend explicitly to both aspects.

There are three ways to ensure correct identification of multiline constructs:

- Add a function to `font-lock-extend-region-functions` that does the *identification* and extends the scan so that the scanned text never starts or ends in the middle of a multiline construct.
- Use the `font-lock-fontify-region-function` hook similarly to extend the scan so that the scanned text never starts or ends in the middle of a multiline construct.
- Somehow identify the multiline construct right when it gets inserted into the buffer (or at any point after that but before font-lock tries to highlight it), and mark it with a `font-lock-multiline` which will instruct font-lock not to start or end the scan in the middle of the construct.

There are 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.

23.6.9.1 Font Lock Multiline

One way to ensure reliable rehighlighting of multiline Font Lock constructs is to put on them the text property `font-lock-multiline`. It should be present and non-`nil` for text that is part of a multiline construct.

When Font Lock is about to highlight a range of text, it first extends the boundaries of the range as necessary so that they do not fall within text marked with the `font-lock-multiline` property. Then it removes any `font-lock-multiline` properties from the range, and highlights it. The highlighting specification (mostly `font-lock-keywords`) must reinstall this property each time, whenever it is appropriate.

Warning: don't use the `font-lock-multiline` property on large ranges of text, because that will make rehighlighting slow.

`font-lock-multiline` [Variable]

If the `font-lock-multiline` variable is set to `t`, Font Lock will try to add the `font-lock-multiline` property automatically on multiline constructs. This is not a universal solution, however, since it slows down Font Lock somewhat. It can miss some multiline constructs, or make the property larger or smaller than necessary.

For elements whose *matcher* is a function, the function should ensure that submatch 0 covers the whole relevant multiline construct, even if only a small subpart will be highlighted. It is often just as easy to add the `font-lock-multiline` property by hand.

The `font-lock-multiline` property is meant to ensure proper refontification; it does not automatically identify new multiline constructs. Identifying the requires that Font Lock mode operate on large enough chunks at a time. This will happen by accident on many cases, which may give the impression that multiline constructs magically work. If you set the `font-lock-multiline` variable non-`nil`, this impression will be even stronger, since the highlighting of those constructs which are found will be properly updated from then on. But that does not work reliably.

To find multiline constructs reliably, you must either manually place the `font-lock-multiline` property on the text before Font Lock mode looks at it, or use `font-lock-fontify-region-function`.

23.6.9.2 Region to Fontify after a Buffer Change

When a buffer is changed, the region that Font Lock refontifies is by default the smallest sequence of whole lines that spans the change. While this works well most of the time, sometimes it doesn't—for example, when a change alters the syntactic meaning of text on an earlier line.

You can enlarge (or even reduce) the region to refontify by setting the following variable:

`font-lock-extend-after-change-region-function` [Variable]

This buffer-local variable is either `nil` or a function for Font Lock mode to call to determine the region to scan and fontify.

The function is given three parameters, the standard *beg*, *end*, and *old-len* from `after-change-functions` (see Section 32.27 [Change Hooks], page 180, vol. 2). 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 Automatic Indentation of code

For programming languages, an important feature of a major mode is to provide automatic indentation. This is controlled in Emacs by `indent-line-function` (see Section 32.17.2 [Mode-Specific Indent], page 151, vol. 2). Writing a good indentation function can be difficult and to a large extent it is still a black art.

Many major mode authors will start by writing a simple indentation function that works for simple cases, for example by comparing with the indentation of the previous text line. For most programming languages that are not really line-based, this tends to scale very poorly: improving such a function to let it handle more diverse situations tends to become more and more difficult, resulting in the end with a large, complex, unmaintainable indentation function which nobody dares to touch.

A good indentation function will usually need to actually parse the text, according to the syntax of the language. Luckily, it is not necessary to parse the text in as much detail as would be needed for a compiler, but on the other hand, the parser embedded in the indentation code will want to be somewhat friendly to syntactically incorrect code.

Good maintainable indentation functions usually fall into two categories: either parsing forward from some "safe" starting point until the position of interest, or parsing backward from the position of interest. Neither of the two is a clearly better choice than the other: parsing backward is often more difficult than parsing forward because programming languages are designed to be parsed forward, but for the purpose of indentation it has the advantage of not needing to guess a "safe" starting point, and it generally enjoys the property that only a minimum of text will be analyzed to decide the indentation of a line, so indentation will tend to be unaffected by syntax errors in some earlier unrelated piece of code. Parsing forward on the other hand is usually easier and has the advantage of making it possible to reindent efficiently a whole region at a time, with a single parse.

Rather than write your own indentation function from scratch, it is often preferable to try and reuse some existing ones or to rely on a generic indentation engine. There are sadly few such engines. The CC-mode indentation code (used with C, C++, Java, Awk and a few other such modes) has been made more generic over the years, so if your language seems somewhat similar to one of those languages, you might try to use that engine. Another one is SMIE which takes an approach in the spirit of Lisp sexps and adapts it to non-Lisp languages.

23.7.1 Simple Minded Indentation Engine

SMIE is a package that provides a generic navigation and indentation engine. Based on a very simple parser using an "operator precedence grammar", it lets major modes extend the sexp-based navigation of Lisp to non-Lisp languages as well as provide a simple to use but reliable auto-indentation.

Operator precedence grammar is a very primitive technology for parsing compared to some of the more common techniques used in compilers. It has the following characteristics: its parsing power is very limited, and it is largely unable to detect syntax errors, but it has the advantage of being algorithmically efficient and able to parse forward just as well as backward. In practice that means that SMIE can use it for indentation based on backward parsing, that it can provide both `forward-sexp` and `backward-sexp` functionality, and that it will naturally work on syntactically incorrect code without any extra effort. The downside is that it also means that most programming languages cannot be parsed correctly using SMIE, at least not without resorting to some special tricks (see Section 23.7.1.5 [SMIE Tricks], page 445).

23.7.1.1 SMIE Setup and Features

SMIE is meant to be a one-stop shop for structural navigation and various other features which rely on the syntactic structure of code, in particular automatic indentation. The main entry point is `smie-setup` which is a function typically called while setting up a major mode.

`smie-setup` *grammar rules-function* **&rest** *keywords* [Function]

Setup SMIE navigation and indentation. *grammar* is a grammar table generated by `smie-prec2->grammar`. *rules-function* is a set of indentation rules for use on `smie-rules-function`. *keywords* are additional arguments, which can include the following keywords:

- `:forward-token` *fun*: Specify the forward lexer to use.
- `:backward-token` *fun*: Specify the backward lexer to use.

Calling this function is sufficient to make commands such as `forward-sexp`, `backward-sexp`, and `transpose-sexps` be able to properly handle structural elements other than just the paired parentheses already handled by syntax tables. For example, if the provided grammar is precise enough, `transpose-sexps` can correctly transpose the two arguments of a `+` operator, taking into account the precedence rules of the language.

Calling 'smie-setup' is also sufficient to make TAB indentation work in the expected way, extends `blink-matching-paren` to apply to elements like `begin...end`, and provides some commands that you can bind in the major mode keymap.

`smie-close-block` [Command]

This command closes the most recently opened (and not yet closed) block.

`smie-down-list` **&optional** *arg* [Command]

This command is like `down-list` but it also pays attention to nesting of tokens other than parentheses, such as `begin...end`.

23.7.1.2 Operator Precedence Grammars

SMIE's precedence grammars simply give to each token a pair of precedences: the left-precedence and the right-precedence. We say `T1 < T2` if the right-precedence of token `T1` is less than the left-precedence of token `T2`. A good way to read this `<` is as a kind of parenthesis: if we find `... T1 something T2 ...` then that should be parsed as `... T1 (something T2 ...` rather than as `... T1 something) T2 ...`. The latter interpretation would be the case if we had `T1 > T2`. If we have `T1 = T2`, it means that token T2 follows token T1 in the same syntactic construction, so typically we have `"begin" = "end"`. Such pairs of precedences are sufficient to express left-associativity or right-associativity of infix operators, nesting of tokens like parentheses and many other cases.

`smie-prec2->grammar` *table* [Function]

This function takes a *prec2* grammar *table* and returns an alist suitable for use in `smie-setup`. The *prec2 table* is itself meant to be built by one of the functions below.

`smie-merge-prec2s` **&rest** *tables* [Function]

This function takes several *prec2 tables* and merges them into a new *prec2* table.

`smie-precs->prec2` *precs* [Function]

This function builds a *prec2* table from a table of precedences *precs*. *precs* should be a list, sorted by precedence (for example `"+"` will come before `"*"`), of elements of the form `(assoc op ...)`, where each *op* is a token that acts as an operator; *assoc* is their associativity, which can be either `left`, `right`, `assoc`, or `nonassoc`. All operators in a given element share the same precedence level and associativity.

`smie-bnf->prec2` *bnf* **&rest** *resolvers* [Function]

This function lets you specify the grammar using a BNF notation. It accepts a *bnf* description of the grammar along with a set of conflict resolution rules *resolvers*, and returns a *prec2* table.

bnf is a list of nonterminal definitions of the form `(nonterm rhs1 rhs2 ...)` where each *rhs* is a (non-empty) list of terminals (aka tokens) or non-terminals.

Not all grammars are accepted:

- An *rhs* cannot be an empty list (an empty list is never needed, since SMIE allows all non-terminals to match the empty string anyway).
- An *rhs* cannot have 2 consecutive non-terminals: each pair of non-terminals needs to be separated by a terminal (aka token). This is a fundamental limitation of operator precedence grammars.

Additionally, conflicts can occur:

- The returned *prec2* table holds constraints between pairs of tokens, and for any given pair only one constraint can be present: T1 < T2, T1 = T2, or T1 > T2.
- A token can be an `opener` (something similar to an open-paren), a `closer` (like a close-paren), or `neither` of the two (e.g. an infix operator, or an inner token like `"else"`).

Precedence conflicts can be resolved via *resolvers*, which is a list of *precs* tables (see `smie-precs->prec2`): for each precedence conflict, if those `precs` tables specify a particular constraint, then the conflict is resolved by using this constraint instead, else a conflict is reported and one of the conflicting constraints is picked arbitrarily and the others are simply ignored.

23.7.1.3 Defining the Grammar of a Language

The usual way to define the SMIE grammar of a language is by defining a new global variable that holds the precedence table by giving a set of BNF rules. For example, the grammar definition for a small Pascal-like language could look like:

```
(require 'smie)
(defvar sample-smie-grammar
  (smie-prec2->grammar
   (smie-bnf->prec2
    '((id)
      (inst ("begin" insts "end")
            ("if" exp "then" inst "else" inst)
            (id ":=" exp)
            (exp))
      (insts (insts ";" insts) (inst))
      (exp (exp "+" exp)
           (exp "*" exp)
           ("(" exps ")"))
      (exps (exps "," exps) (exp)))
    '((assoc ";"))
    '((assoc ","))
    '((assoc "+") (assoc "*")))))
```

A few things to note:

- The above grammar does not explicitly mention the syntax of function calls: SMIE will automatically allow any sequence of sexps, such as identifiers, balanced parentheses, or `begin ... end` blocks to appear anywhere anyway.
- The grammar category `id` has no right hand side: this does not mean that it can match only the empty string, since as mentioned any sequence of sexps can appear anywhere anyway.
- Because non terminals cannot appear consecutively in the BNF grammar, it is difficult to correctly handle tokens that act as terminators, so the above grammar treats `";"` as a statement *separator* instead, which SMIE can handle very well.
- Separators used in sequences (such as `","` and `";"` above) are best defined with BNF rules such as `(foo (foo "separator" foo) ...)` which generate precedence conflicts which are then resolved by giving them an explicit `(assoc "separator")`.
- The `("(" exps ")")` rule was not needed to pair up parens, since SMIE will pair up any characters that are marked as having paren syntax in the syntax table. What this rule does instead (together with the definition of `exps`) is to make it clear that `","` should not appear outside of parentheses.
- Rather than have a single *precs* table to resolve conflicts, it is preferable to have several tables, so as to let the BNF part of the grammar specify relative precedences where possible.
- Unless there is a very good reason to prefer `left` or `right`, it is usually preferable to mark operators as associative, using `assoc`. For that reason `"+"` and `"*"` are defined above as `assoc`, although the language defines them formally as left associative.

23.7.1.4 Defining Tokens

SMIE comes with a predefined lexical analyzer which uses syntax tables in the following way: any sequence of characters that have word or symbol syntax is considered a token, and so is any sequence of characters that have punctuation syntax. This default lexer is often a good starting point but is rarely actually correct for any given language. For example, it will consider `"2,+3"` to be composed of 3 tokens: `"2"`, `",+"`, and `"3"`.

To describe the lexing rules of your language to SMIE, you need 2 functions, one to fetch the next token, and another to fetch the previous token. Those functions will usually first skip whitespace and comments and then look at the next chunk of text to see if it is a special token. If so it should skip the token and return a description of this token. Usually this is simply the string extracted from the buffer, but it can be anything you want. For example:

```
(defvar sample-keywords-regexp
  (regexp-opt '("+" "*" "," ";" ">" ">=" "<" "<=" ":=" "=")))
```

```
(defun sample-smie-forward-token ()
  (forward-comment (point-max))
  (cond
   ((looking-at sample-keywords-regexp)
    (goto-char (match-end 0))
    (match-string-no-properties 0))
   (t (buffer-substring-no-properties
       (point)
       (progn (skip-syntax-forward "w_")
              (point))))))
(defun sample-smie-backward-token ()
  (forward-comment (- (point)))
  (cond
   ((looking-back sample-keywords-regexp (- (point) 2) t)
    (goto-char (match-beginning 0))
    (match-string-no-properties 0))
   (t (buffer-substring-no-properties
       (point)
       (progn (skip-syntax-backward "w_")
              (point))))))
```

Notice how those lexers return the empty string when in front of parentheses. This is because SMIE automatically takes care of the parentheses defined in the syntax table. More specifically if the lexer returns nil or an empty string, SMIE tries to handle the corresponding text as a sexp according to syntax tables.

23.7.1.5 Living With a Weak Parser

The parsing technique used by SMIE does not allow tokens to behave differently in different contexts. For most programming languages, this manifests itself by precedence conflicts when converting the BNF grammar.

Sometimes, those conflicts can be worked around by expressing the grammar slightly differently. For example, for Modula-2 it might seem natural to have a BNF grammar that looks like this:

```
...
(inst ("IF" exp "THEN" insts "ELSE" insts "END")
      ("CASE" exp "OF" cases "END")
      ...)
(cases (cases "|" cases)
       (caselabel ":" insts)
       ("ELSE" insts))
...
```

But this will create conflicts for `"ELSE"`: on the one hand, the IF rule implies (among many other things) that `"ELSE" = "END"`; but on the other hand, since `"ELSE"` appears within `cases`, which appears left of `"END"`, we also have `"ELSE" > "END"`. We can solve the conflict either by using:

```
...
(inst ("IF" exp "THEN" insts "ELSE" insts "END")
      ("CASE" exp "OF" cases "END")
      ("CASE" exp "OF" cases "ELSE" insts "END")
      ...)
(cases (cases "|" cases) (caselabel ":" insts))
...
```

or

```
     ...
     (inst ("IF" exp "THEN" else "END")
           ("CASE" exp "OF" cases "END")
           ...)
     (else (insts "ELSE" insts))
     (cases (cases "|" cases) (caselabel ":" insts) (else))
     ...
```

Reworking the grammar to try and solve conflicts has its downsides, tho, because SMIE assumes that the grammar reflects the logical structure of the code, so it is preferable to keep the BNF closer to the intended abstract syntax tree.

Other times, after careful consideration you may conclude that those conflicts are not serious and simply resolve them via the *resolvers* argument of `smie-bnf->prec2`. Usually this is because the grammar is simply ambiguous: the conflict does not affect the set of programs described by the grammar, but only the way those programs are parsed. This is typically the case for separators and associative infix operators, where you want to add a resolver like `'((assoc "|"))`. Another case where this can happen is for the classic *dangling else* problem, where you will use `'((assoc "else" "then"))`. It can also happen for cases where the conflict is real and cannot really be resolved, but it is unlikely to pose a problem in practice.

Finally, in many cases some conflicts will remain despite all efforts to restructure the grammar. Do not despair: while the parser cannot be made more clever, you can make the lexer as smart as you want. So, the solution is then to look at the tokens involved in the conflict and to split one of those tokens into 2 (or more) different tokens. E.g. if the grammar needs to distinguish between two incompatible uses of the token `"begin"`, make the lexer return different tokens (say `"begin-fun"` and `"begin-plain"`) depending on which kind of `"begin"` it finds. This pushes the work of distinguishing the different cases to the lexer, which will thus have to look at the surrounding text to find ad-hoc clues.

23.7.1.6 Specifying Indentation Rules

Based on the provided grammar, SMIE will be able to provide automatic indentation without any extra effort. But in practice, this default indentation style will probably not be good enough. You will want to tweak it in many different cases.

SMIE indentation is based on the idea that indentation rules should be as local as possible. To this end, it relies on the idea of *virtual* indentation, which is the indentation that a particular program point would have if it were at the beginning of a line. Of course, if that program point is indeed at the beginning of a line, its virtual indentation is its current indentation. But if not, then SMIE uses the indentation algorithm to compute the virtual indentation of that point. Now in practice, the virtual indentation of a program point does not have to be identical to the indentation it would have if we inserted a newline before it.

To see how this works, the SMIE rule for indentation after a `{` in C does not care whether the `{` is standing on a line of its own or is at the end of the preceding line. Instead, these different cases are handled in the indentation rule that decides how to indent before a `{`.

Another important concept is the notion of *parent*: The *parent* of a token, is the head token of the nearest enclosing syntactic construct. For example, the parent of an `else` is the `if` to which it belongs, and the parent of an `if`, in turn, is the lead token of the surrounding construct. The command `backward-sexp` jumps from a token to its parent, but there are some caveats: for *openers* (tokens which start a construct, like `if`), you need to start with point before the token, while for others you need to start with point after the token. `backward-sexp` stops with point before the parent token if that is the *opener* of the token of interest, and otherwise it stops with point after the parent token.

SMIE indentation rules are specified using a function that takes two arguments *method* and *arg* where the meaning of *arg* and the expected return value depend on *method*.

method can be:

- `:after`, in which case *arg* is a token and the function should return the *offset* to use for indentation after *arg*.
- `:before`, in which case *arg* is a token and the function should return the *offset* to use to indent *arg* itself.
- `:elem`, in which case the function should return either the offset to use to indent function arguments (if *arg* is the symbol `arg`) or the basic indentation step (if *arg* is the symbol `basic`).
- `:list-intro`, in which case *arg* is a token and the function should return non-`nil` if the token is followed by a list of expressions (not separated by any token) rather than an expression.

When *arg* is a token, the function is called with point just before that token. A return value of nil always means to fallback on the default behavior, so the function should return nil for arguments it does not expect.

offset can be:

- `nil`: use the default indentation rule.
- `(column . column)`: indent to column *column*.
- *number*: offset by *number*, relative to a base token which is the current token for `:after` and its parent for `:before`.

23.7.1.7 Helper Functions for Indentation Rules

SMIE provides various functions designed specifically for use in the indentation rules function (several of those functions break if used in another context). These functions all start with the prefix `smie-rule-`.

`smie-rule-bolp` [Function]

Return non-`nil` if the current token is the first on the line.

`smie-rule-hanging-p` [Function]

Return non-`nil` if the current token is *hanging*. A token is *hanging* if it is the last token on the line and if it is preceded by other tokens: a lone token on a line is not hanging.

`smie-rule-next-p` **&rest** *tokens* [Function]

Return non-`nil` if the next token is among *tokens*.

`smie-rule-prev-p` **&rest** *tokens* [Function]

Return non-`nil` if the previous token is among *tokens*.

`smie-rule-parent-p` **&rest** *parents* [Function]

Return non-`nil` if the current token's parent is among *parents*.

`smie-rule-sibling-p` [Function]

Return non-`nil` if the current token's parent is actually a sibling. This is the case for example when the parent of a `","` is just the previous `","`.

`smie-rule-parent` **&optional** *offset* [Function]

Return the proper offset to align the current token with the parent. If non-`nil`, *offset* should be an integer giving an additional offset to apply.

`smie-rule-separator` *method* [Function]

Indent current token as a *separator*.

By *separator*, we mean here a token whose sole purpose is to separate various elements within some enclosing syntactic construct, and which does not have any semantic significance in itself (i.e. it would typically not exist as a node in an abstract syntax tree).

Such a token is expected to have an associative syntax and be closely tied to its syntactic parent. Typical examples are `","` in lists of arguments (enclosed inside parentheses), or `";"` in sequences of instructions (enclosed in a `{...}` or `begin...end` block).

method should be the method name that was passed to 'smie-rules-function'.

23.7.1.8 Sample Indentation Rules

Here is an example of an indentation function:

```
(defun sample-smie-rules (kind token)
  (pcase (cons kind token)
    (`(:elem . basic) sample-indent-basic)
    (`(,_ . ",") (smie-rule-separator kind))
    (`(:after . ":=") sample-indent-basic)
    (`(:before . ,(or `"begin" `"(" `"{"))
     (if (smie-rule-hanging-p) (smie-rule-parent)))
    (`(:before . "if")
     (and (not (smie-rule-bolp)) (smie-rule-prev-p "else")
          (smie-rule-parent)))))
```

A few things to note:

- The first case indicates the basic indentation increment to use. If `sample-indent-basic` is nil, then SMIE uses the global setting `smie-indent-basic`. The major mode could have set `smie-indent-basic` buffer-locally instead, but that is discouraged.
- The rule for the token `","` make SMIE try to be more clever when the comma separator is placed at the beginning of lines. It tries to outdent the separator so as to align the code after the comma; for example:

```
        x = longfunctionname (
                arg1
              , arg2
            );
```

- The rule for indentation after `":="` exists because otherwise SMIE would treat `":="` as an infix operator and would align the right argument with the left one.
- The rule for indentation before `"begin"` is an example of the use of virtual indentation: This rule is used only when `"begin"` is hanging, which can happen only when `"begin"` is not at the beginning of a line. So this is not used when indenting `"begin"` itself but only when indenting something relative to this `"begin"`. Concretely, this rule changes the indentation from:

  ```
  if x > 0 then begin
          dosomething(x);
      end
  ```

  to

  ```
  if x > 0 then begin
      dosomething(x);
  end
  ```

- The rule for indentation before `"if"` is similar to the one for `"begin"`, but where the purpose is to treat `"else if"` as a single unit, so as to align a sequence of tests rather than indent each test further to the right. This function does this only in the case where the `"if"` is not placed on a separate line, hence the `smie-rule-bolp` test.

  If we know that the `"else"` is always aligned with its `"if"` and is always at the beginning of a line, we can use a more efficient rule:

  ```
  ((equal token "if")
   (and (not (smie-rule-bolp))
        (smie-rule-prev-p "else")
        (save-excursion
          (sample-smie-backward-token)
          (cons 'column (current-column)))))
  ```

  The advantage of this formulation is that it reuses the indentation of the previous `"else"`, rather than going all the way back to the first `"if"` of the sequence.

23.8 Desktop Save Mode

Desktop Save Mode is a feature to save the state of Emacs from one session to another. The user-level commands for using Desktop Save Mode are described in the GNU Emacs Manual (see Section "Saving Emacs Sessions" in *the GNU Emacs Manual*). Modes whose buffers visit a file, don't have to do anything to use this feature.

For buffers not visiting a file to have their state saved, the major mode must bind the buffer local variable `desktop-save-buffer` to a non-`nil` value.

`desktop-save-buffer` [Variable]

If this buffer-local variable is non-`nil`, the buffer will have its state saved in the desktop file at desktop save. If the value is a function, it is called at desktop save with argument *desktop-dirname*, and its value is saved in the desktop file along with the state of the buffer for which it was called. When file names are returned as part of the auxiliary information, they should be formatted using the call

```
(desktop-file-name file-name desktop-dirname)
```

For buffers not visiting a file to be restored, the major mode must define a function to do the job, and that function must be listed in the alist `desktop-buffer-mode-handlers`.

`desktop-buffer-mode-handlers` [Variable]

Alist with elements

```
(major-mode . restore-buffer-function)
```

The function *restore-buffer-function* will be called with argument list

```
(buffer-file-name buffer-name desktop-buffer-misc)
```

and it should return the restored buffer. Here *desktop-buffer-misc* is the value returned by the function optionally bound to `desktop-save-buffer`.

24 Documentation

GNU Emacs has convenient built-in help facilities, most of which derive their information from documentation strings associated with functions and variables. This chapter describes how to access documentation strings in Lisp programs. See Section D.6 [Documentation Tips], page 450, vol. 2, for how to write good documentation strings.

Note that the documentation strings for Emacs are not the same thing as the Emacs manual. Manuals have their own source files, written in the Texinfo language; documentation strings are specified in the definitions of the functions and variables they apply to. A collection of documentation strings is not sufficient as a manual because a good manual is not organized in that fashion; it is organized in terms of topics of discussion.

For commands to display documentation strings, see Section "Help" in *The GNU Emacs Manual*.

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) that briefly describes what the function or variable does. 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 documentation strings; see Section D.6 [Documentation Tips], page 450, vol. 2.

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 454.)

Emacs Lisp mode fills documentation strings to the width specified by `emacs-lisp-docstring-fill-column`.

Exactly where a documentation string is stored depends on how its function or variable was defined or loaded into memory:

- When you define a function (see Section 12.2 [Lambda Expressions], page 165, and see Section 12.2.4 [Function Documentation], page 167), the documentation string is stored in the function definition itself. You can also put function documentation in the `function-documentation` property of a function name. That is useful for function definitions which can't hold a documentation string, such as keyboard macros.
- When you define a variable with a `defvar` or related form (see Section 11.5 [Defining Variables], page 141), the documentation is stored in the variable's `variable-documentation` property.

- To save memory, the documentation for preloaded functions and variables (including primitive functions and autoloaded functions) is not kept in memory, but in the file '`emacs/etc/DOC-`*`version`*', where *version* is the Emacs version number (see Section 1.4 [Version Info], page 6).
- When a function or variable is loaded from a byte-compiled file during the Emacs session, its documentation string is not loaded into memory. Instead, Emacs looks it up in the byte-compiled file as needed. See Section 16.3 [Docs and Compilation], page 226.

Regardless of where the documentation string is stored, you can retrieve it using the `documentation` or `documentation-property` function, described in the next section.

24.2 Access to Documentation Strings

`documentation-property` *symbol property* **&optional** *verbatim* [Function]

This function returns the documentation string recorded in *symbol*'s property list under property *property*. It is most often used to look up the documentation strings of variables, for which *property* is `variable-documentation`. However, it can also be used to look up other kinds of documentation, such as for customization groups (but for function documentation, use the `documentation` command, below).

If the value recorded in the property list refers to a documentation string stored in a '`DOC-`*`version`*' file or a byte-compiled file, it looks up that string and returns it. If the property value isn't `nil`, isn't a string, and doesn't refer to text in a file, then it is evaluated as a Lisp expression to obtain a string.

The last thing this function does is pass the string through `substitute-command-keys` to substitute actual key bindings (see Section 24.3 [Keys in Documentation], page 454). However, it skips this step if *verbatim* is non-`nil`.

```
(documentation-property 'command-line-processed
   'variable-documentation)
     ⇒ "Non-nil once command line has been processed"
(symbol-plist 'command-line-processed)
     ⇒ (variable-documentation 188902)
(documentation-property 'emacs 'group-documentation)
     ⇒ "Customization of the One True Editor."
```

`documentation` *function* **&optional** *verbatim* [Function]

This function returns the documentation string of *function*. It handles macros, named keyboard macros, and special forms, as well as ordinary functions.

If *function* is a symbol, this function first looks for the `function-documentation` property of that symbol; if that has a non-`nil` value, the documentation comes from that value (if the value is not a string, it is evaluated). If *function* is not a symbol, or if it has no `function-documentation` property, then `documentation` extracts the documentation string from the actual function definition, reading it from a file if called for.

Finally, unless *verbatim* is non-`nil`, 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.
    (help-setup-xref (list 'describe-symbols pattern) (interactive-p))
    (with-help-window (help-buffer)
      (mapcar describe-func (sort sym-list 'string<)))))
```

The `describe-symbols` function works like `apropos`, but provides more information.

```
(describe-symbols "goal")

---------- Buffer: *Help* ----------
goal-column     Option
Semipermanent goal column for vertical motion, as set by ...

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* ----------
```

`Snarf-documentation` *filename* [Function]

This function is used when building Emacs, just before the runnable Emacs is dumped. It finds the positions of the documentation strings stored in the file *filename*, and records those positions into memory in the function definitions and variable property lists. See Section E.1 [Building Emacs], page 457, vol. 2.

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 459.

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.

If a command has multiple bindings, this function normally uses the first one it finds. You can specify one particular key binding by assigning an `:advertised-binding` symbol property to the command, like this:

```
(put 'undo :advertised-binding [?\C-/])
```

The `:advertised-binding` property also affects the binding shown in menu items (see Section 22.17.5 [Menu Bar], page 391). The property is ignored if it specifies a key binding that the command does not actually have.

Here are examples of the special sequences:

```
(substitute-command-keys
   "To abort recursive edit, type: \\[abort-recursive-edit]")
⇒ "To abort recursive edit, type: C-]"

(substitute-command-keys
   "The keys that are defined for the minibuffer here are:
  \\{minibuffer-local-must-match-map}")
⇒ "The keys that are defined for the minibuffer here are:

?               minibuffer-completion-help
SPC             minibuffer-complete-word
TAB             minibuffer-complete
C-j             minibuffer-complete-and-exit
RET             minibuffer-complete-and-exit
C-g             abort-recursive-edit
"

(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 450, vol. 2, 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 21.7 [Input Events], page 327, for more information about valid events.

```
(key-description [?\M-3 delete])
     ⇒ "M-3 <delete>"
(key-description [delete] "\M-3")
     ⇒ "M-3 <delete>"
```

See also the examples for `single-key-description`, below.

`single-key-description` *event* **&optional** *no-angles* [Function]

This function returns a string describing *event* in the standard Emacs notation for keyboard input. A normal printing character appears as itself, but a control character turns into a string starting with '`C-`', a meta character turns into a string starting with '`M-`', and space, tab, etc. appear as '`SPC`', '`TAB`', etc. A function key symbol appears inside angle brackets '`<...>`'. An event that is a list appears as the name of the symbol in the CAR of the list, inside angle brackets.

If the optional argument *no-angles* is non-`nil`, the angle brackets around function keys and event symbols are omitted; this is for compatibility with old versions of Emacs which didn't use the brackets.

```
(single-key-description ?\C-x)
     ⇒ "C-x"
(key-description "\C-x \M-y \n \t \r \f123")
     ⇒ "C-x SPC M-y SPC C-j SPC TAB SPC RET SPC C-l 1 2 3"
(single-key-description 'delete)
     ⇒ "<delete>"
(single-key-description 'C-mouse-1)
     ⇒ "<C-mouse-1>"
(single-key-description 'C-mouse-1 t)
     ⇒ "C-mouse-1"
```

`text-char-description` *character* [Function]

This function returns a string describing *character* in the standard Emacs notation for characters that 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* [Command]

This function is used mainly for operating on keyboard macros, but it can also be used as a rough inverse for `key-description`. You call it with a string containing key descriptions, separated by spaces; it returns a string or vector containing the corresponding events. (This may or may not be a single valid key sequence, depending on what events you use; see Section 22.1 [Key Sequences], page 360.) 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 function-doc variable-doc
 plist-doc widget-doc face-doc group-doc)
```

Here, *score* is an integer measure of how important the symbol seems to be as a match. Each of the remaining elements is a documentation string, or `nil`, for *symbol* as a function, variable, etc.

It also displays the symbols in a buffer named '`*Apropos*`', each with a one-line description taken from the beginning of its documentation string.

If *do-all* is non-`nil`, or if the user option `apropos-do-all` is non-`nil`, then `apropos` also shows key bindings for the functions that are found; it also shows *all* interned symbols, not just meaningful ones (and it lists them in the return value as well).

`help-map` [Variable]

The value of this variable is a local keymap for characters following the Help key, `C-h`.

`help-command` [Prefix Command]

This symbol is not a function; its function definition cell holds the keymap known as `help-map`. It is defined in '`help.el`' as follows:

```
(define-key global-map (string help-char) 'help-command)
(fset 'help-command help-map)
```

`help-char` [User Option]

The value of this variable is the help character—the character that Emacs recognizes as meaning Help. By default, its value is 8, which stands for *C-h*. When Emacs reads this character, if `help-form` is a non-`nil` Lisp expression, it evaluates that expression, and displays the result in a window if it is a string.

Usually the value of `help-form` is `nil`. Then the help character has no special meaning at the level of command input, and it becomes part of a key sequence in the normal way. The standard key binding of *C-h* is a prefix key for several general-purpose help features.

The help character is special after prefix keys, too. If it has no binding as a subcommand of the prefix key, it runs `describe-prefix-bindings`, which displays a list of all the subcommands of the prefix key.

`help-event-list` [User Option]

The value of this variable is a list of event types that serve as alternative "help characters". These events are handled just like the event specified by `help-char`.

`help-form` [Variable]

If this variable is non-`nil`, its value is a form to evaluate whenever the character `help-char` is read. If evaluating the form produces a string, that string is displayed.

A command that calls `read-event`, `read-char-choice`, 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 313).

`prefix-help-command` [Variable]

This variable holds a function to print help for a prefix key. The function is called when the user types a prefix key followed by the help character, and the help character has no binding after that prefix. The variable's default value is `describe-prefix-bindings`.

`describe-prefix-bindings` [Command]

This function calls `describe-bindings` to display a list of all the subcommands of the prefix key of the most recent key sequence. The prefix described consists of all but the last event of that key sequence. (The last event is, presumably, the help character.)

The following two functions are meant for modes that want to provide help without relinquishing control, such as the "electric" modes. Their names begin with '`Helper`' to distinguish them from the ordinary help functions.

`Helper-describe-bindings` [Command]

This command pops up a window displaying a help buffer containing a listing of all of the key bindings from both the local and global keymaps. It works by calling `describe-bindings`.

`Helper-help` [Command]

This command provides help for the current mode. It prompts the user in the minibuffer with the message '`Help (Type ? for further options)`', and then provides assistance in finding out what the key bindings are, and what the mode is intended for. It returns `nil`.

This can be customized by changing the map `Helper-help-map`.

`data-directory` [Variable]

This variable holds the name of the directory in which Emacs finds certain documentation and text files that come with Emacs.

`help-buffer` [Function]

This function returns the name of the help buffer, which is normally '`*Help*`'; if such a buffer does not exist, it is first created.

`with-help-window` *buffer-name body...* [Macro]

This macro evaluates the *body* forms, inserting any output they produce into a buffer named *buffer-name* like `with-output-to-temp-buffer` (see Section 38.8 [Temporary Displays], page 313, vol. 2). (Usually, *buffer-name* should be the value returned by the function `help-buffer`.) It also puts the specified buffer into Help mode and displays a message telling the user how to quit and scroll the help window.

`help-setup-xref` *item interactive-p* [Function]

This function updates the cross reference data in the '`*Help*`' buffer, which is used to regenerate the help information when the user clicks on the '`Back`' or '`Forward`' buttons. Most commands that use the '`*Help*`' buffer should invoke this function before clearing the buffer. The *item* argument should have the form `(`*`function`*` . `*`args`*`)`, where *function* is a function to call, with argument list *args*, to regenerate the help buffer. The *interactive-p* argument is non-`nil` if the calling command was invoked interactively; in that case, the stack of items for the '`*Help*`' buffer's '`Back`' buttons is cleared.

See [describe-symbols example], page 453, for an example of using `help-buffer`, `with-help-window`, and `help-setup-xref`.

`make-help-screen` *fname help-line help-text help-map* [Macro]

This macro defines a help command named *fname* that acts like a prefix key that shows a list of the subcommands it offers.

When invoked, *fname* displays *help-text* in a window, then reads and executes a key sequence according to *help-map*. The string *help-text* should describe the bindings available in *help-map*.

The command *fname* is defined to handle a few events itself, by scrolling the display of *help-text*. When *fname* reads one of those special events, it does the scrolling and then reads another event. When it reads an event that is not one of those few, and which has a binding in *help-map*, it executes that key's binding and then returns.

The argument *help-line* should be a single-line summary of the alternatives in *help-map*. In the current version of Emacs, this argument is used only if you set the option `three-step-help` to `t`.

This macro is used in the command `help-for-help` which is the binding of *`C-h C-h`*.

`three-step-help` [User Option]

If this variable is non-`nil`, commands defined with `make-help-screen` display their *help-line* strings in the echo area at first, and display the longer *help-text* strings only if the user types the help character again.

25 Files

This chapter describes the Emacs Lisp functions and variables to find, create, view, save, and otherwise work with files and file directories. A few other file-related functions are described in Chapter 27 [Buffers], page 1, vol. 2, and those related to backups and auto-saving are described in Chapter 26 [Backups and Auto-Saving], page 502.

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 '`../`'). See Section 25.8.4 [File Name Expansion], page 486.

In addition, certain *magic* file names are handled specially. For example, when a remote file name is specified, Emacs accesses the file over the network via an appropriate protocol (see Section "Remote Files" in *The GNU Emacs Manual*). This handling is done at a very low level, so you may assume that all the functions described in this chapter accept magic file names as file name arguments, except where noted. See Section 25.11 [Magic File Names], page 493, for details.

When file I/O functions signal Lisp errors, they usually use the condition `file-error` (see Section 10.5.3.3 [Handling Errors], page 130). 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 33.11 [Locales], page 207, vol. 2).

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 5, vol. 2, 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 467.

`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 28.10 [Switching Buffers], page 37, vol. 2.)

If *wildcards* is non-`nil`, which is always true in an interactive call, then `find-file` expands wildcard characters in *filename* and visits all the matching files.

When `find-file` is called interactively, it prompts for *filename* in the minibuffer.

`find-file-literally` *filename* [Command]

This command visits *filename*, like `find-file` does, but it does not perform any format conversions (see Section 25.12 [Format Conversion], page 497), character code conversions (see Section 33.9 [Coding Systems], page 193, vol. 2), or end-of-line conversions (see Section 33.9.1 [Coding System Basics], page 193, vol. 2). The buffer visiting the file is made unibyte, and its major mode is Fundamental mode, regardless of the file name. File local variable specifications in the file (see Section 11.11 [File Local Variables], page 156) are ignored, and automatic decompression and adding a newline at the end of the file due to `require-final-newline` (see Section 25.2 [Saving Buffers], page 465) are also disabled.

Note that if Emacs already has a buffer visiting the same file non-literally, it will not visit the same file literally, but instead just switch to the existing buffer. If you want to be sure of accessing a file's contents literally, you should create a temporary buffer and then read the file contents into it using `insert-file-contents-literally` (see Section 25.3 [Reading from Files], page 467).

`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 33.9 [Coding Systems], page 193, vol. 2), including end-of-line conversion, and format conversion (see Section 25.12 [Format Conversion], page 497). 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 464). 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 28.10 [Switching Buffers], page 37, vol. 2.

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 9, vol. 2, for related functions and variables.

When this command is called interactively, it prompts for *filename*.

`find-file-wildcards` [User Option]

If this variable is non-`nil`, then the various `find-file` commands check for wildcard characters and visit all the files that match them (when invoked interactively or when their *wildcards* argument is non-`nil`). If this option is `nil`, then the `find-file` commands ignore their *wildcards* argument and never treat wildcard characters specially.

`find-file-hook` [User Option]

The value of this variable is a list of functions to be called after a file is visited. The file's local-variables specification (if any) will have been processed before the hooks are run. The buffer visiting the file is current when the hook functions are run.

This variable is a normal hook. See Section 23.1 [Hooks], page 396.

`find-file-not-found-functions` [Variable]

The value of this variable is a list of functions to be called when `find-file` or `find-file-noselect` is passed a nonexistent file name. `find-file-noselect` calls these functions as soon as it detects a nonexistent file. It calls them in the order of the list, until one of them returns non-`nil`. `buffer-file-name` is already set up.

This is not a normal hook because the values of the functions are used, and in many cases only some of the functions are called.

`find-file-literally` [Variable]

This buffer-local variable, if set to a non-`nil` value, makes `save-buffer` behave as if the buffer were visiting its file literally, i.e. without conversions of any kind. The command `find-file-literally` sets this variable's local value, but other equivalent functions and commands can do that as well, e.g. to avoid automatic addition of a newline at the end of the file. This variable is permanent local, so it is unaffected by changes of major modes.

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 13, vol. 2.

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 13, vol. 2).

`after-find-file` **&optional** *error warn noauto after-find-file-from-revert-buffer nomodes* [Function]

This function sets the buffer major mode, and parses local variables (see Section 23.2.2 [Auto Major Mode], page 403). It is called by `find-file-noselect` and by the default revert function (see Section 26.3 [Reverting], page 510).

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 13, vol. 2). 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 5, vol. 2) 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 497).

`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 502.

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 33.9.3 [Lisp and Coding Systems], page 195, vol. 2), perform the encoding (see Section 33.9.7 [Explicit Encoding], page 203, vol. 2), and set `last-coding-system-used` to the coding system that was used (see Section 33.9.2 [Encoding and I/O], page 194, vol. 2).

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 396.

`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 504. Yet, at the same time, saving a precious file always breaks all hard links between the file you save and other file names.

Some modes give this variable a non-`nil` buffer-local value in particular buffers.

require-final-newline [User Option]

This variable determines whether files may be written out that do *not* end with a newline. If the value of the variable is `t`, then `save-buffer` silently adds a newline at the end of the buffer whenever it does not already end in one. If the value is `visit`, Emacs adds a missing newline just after it visits the file. If the value is `visit-save`, Emacs adds a missing newline both on visiting and on saving. For any other non-`nil` value, `save-buffer` asks the user whether to add a newline each time the case arises.

If the value of the variable is `nil`, then `save-buffer` doesn't add newlines at all. `nil` is the default value, but a few major modes set it to `t` in particular buffers.

See also the function `set-visited-file-name` (see Section 27.4 [Buffer File Name], page 5, vol. 2).

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.

This function checks the file contents against the defined file formats, and converts the file contents if appropriate and also calls the functions in the list `after-insert-file-functions`. See Section 25.12 [Format Conversion], page 497. Normally, one of the functions in the `after-insert-file-functions` list determines the coding system (see Section 33.9 [Coding Systems], page 193, vol. 2) used for decoding the file's contents, including end-of-line conversion. However, if the file contains null bytes, it is by default visited without any code conversions. See Section 33.9.3 [Lisp and Coding Systems], page 195, vol. 2.

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 replace* [Function]

This function works like `insert-file-contents` except that it does not run `find-file-hook`, and does not do format decoding, character code conversion, automatic uncompression, and so on.

If you want to pass a file name to another process so that another program can read the file, use the function `file-local-copy`; see Section 25.11 [Magic File Names], page 493.

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` and also calls the functions in the list `write-region-annotate-functions`. See Section 25.12 [Format Conversion], page 497.

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 126).

See also `with-temp-buffer` in [The Current Buffer], page 3, vol. 2.

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 8, vol. 2.

`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 permissions 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 475.

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).

`file-equal-p` *file1 file2* [Function]

This function returns `t` if the files *file1* and *file2* name the same file. If *file1* or *file2* does not exist, the return value is unspecified.

`file-in-directory-p` *file dir* [Function]

This function returns `t` if *file* is a file in directory *dir*, or in a subdirectory of *dir*. It also returns `t` if *file* and *dir* are the same directory. It compares the `file-truename` values of the two directories (see Section 25.6.3 [Truenames], page 474). If *dir* does not name an existing directory, the return value is `nil`.

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]

This function returns the truename of the file *filename*. If the argument is not an absolute file name, this function first expands it against `default-directory`.

This function does not expand environment variables. Only `substitute-in-file-name` does that. See [Definition of substitute-in-file-name], page 487.

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 486.

`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 5, vol. 2, 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 permissions, 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* describing the *file permissions* of *filename*, as an integer. It recursively follows symbolic links in *filename* at all levels. If *filename* does not exist, the return value is `nil`.

See Section "File Permissions" in *The* GNU `Coreutils` *Manual*, for a description of mode bits. 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.

```
(file-modes "~/junk/diffs")
     ⇒ 492               ; Decimal integer.
(format "%o" 492)
     ⇒ "754"             ; Convert to octal.

(set-file-modes "~/junk/diffs" #o666)
     ⇒ nil

% ls -l diffs
  -rw-rw-rw-  1 lewis 0 3063 Oct 30 16:00 diffs
```

See Section 25.7 [Changing Files], page 479, for functions that change file permissions, such as `set-file-modes`.

MS-DOS note: On MS-DOS, there is no such thing as an "executable" file mode bit. So `file-modes` considers a file executable if its name ends in one of the standard executable extensions, such as '`.com`', '`.bat`', '`.exe`', and some others. Files that begin with the Unix-standard '`#!`' signature, such as shell and Perl scripts, are also considered executable. Directories are also reported as executable, for compatibility with Unix. These conventions are also followed by `file-attributes`, below.

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 functions 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 479).
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 399, vol. 2.) Note that on some FAT-based filesystems, only the date of last access is recorded, so this time will always hold the midnight of the day of last access.
5. The time of last modification as a list of two integers (as above). This is the last time when the file's contents were modified.
6. The time of last status change as a list of two integers (as above). This is the time of the last change to the file's access mode bits, its owner and group, and other information recorded in the filesystem for the file, beyond the file's contents.
7. The size of the file in bytes. 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 but dividing it by $2^1$6 yields a representable integer, then the value has the form `(`*`high`*` . `*`low`*`)`, where *low* holds the low 16 bits. If the inode number is too wide for even that, the value is of the form `(`*`high middle`*` . `*`low`*`)`, where `high` holds the high bits, *middle* the middle 24 bits, and *low* the low 16 bits.
11. The filesystem number of the device that the file is on. 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"
          (19145 42977)
          (19141 59576)
          (18340 17300)
          122295 "-rw-rw-rw-"
          nil  (5888 2 . 43978)
          (15479 . 46724))
```

and here is how the result is interpreted:

`nil` is neither a directory nor a symbolic link.

`1` has only one name (the name '`files.texi`' in the current default directory).

`"lh"` is owned by the user with name "lh".

`"users"` is in the group with name "users".

`(19145 42977)`
was last accessed on Oct 5 2009, at 10:01:37.

`(19141 59576)`
last had its contents modified on Oct 2 2009, at 13:49:12.

`(18340 17300)`
last had its status changed on Feb 2 2008, at 12:19:00.

`122295` is 122295 bytes long. (It may not contain 122295 characters, though, if some of the bytes belong to multibyte sequences, and also if the end-of-line format is CR-LF.)

`"-rw-rw-rw-"`
has a mode of read and write access for the owner, group, and world.

`nil` would retain the same GID if it were recreated.

`(5888 2 . 43978)`
has an inode number of 6473924464520138.

`(15479 . 46724)`
is on the file-system device whose number is 1014478468.

SELinux is a Linux kernel feature which provides more sophisticated file access controls than ordinary "Unix-style" file permissions. If Emacs has been compiled with SELinux support on a system with SELinux enabled, you can use the function `file-selinux-context` to retrieve a file's SELinux security context. For the function `set-file-selinux-context`, see Section 25.7 [Changing Files], page 479.

`file-selinux-context` *filename* [Function]

This function returns the SELinux security context of the file *filename*. This return value is a list of the form `(user role type range)`, whose elements are the context's user, role, type, and range respectively, as Lisp strings. See the SELinux documentation for details about what these actually mean.

If the file does not exist or is inaccessible, or if the system does not support SELinux, or if Emacs was not compiled with SELinux support, then the return value is `(nil nil nil nil)`.

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*), or for an executable file in the standard list of executable file directories.

To search for a user-specific configuration file, See Section 25.8.7 [Standard File Names], page 490, for the `locate-user-emacs-file` function.

`locate-file` *filename path* **&optional** *suffixes predicate* [Function]

This function searches for a file whose name is *filename* in a list of directories given by *path*, trying the suffixes in *suffixes*. If it finds such a file, it returns the file's absolute file name (see Section 25.8.2 [Relative File Names], page 484); 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 257, vol. 2), `load-suffixes`, `load-file-rep-suffixes` and the return value of the function `get-load-suffixes` (see Section 15.2 [Load Suffixes], page 211).

Typical values for *path* are `exec-path` (see Section 37.1 [Subprocess Creation], page 257, vol. 2) when looking for executable programs, or `load-path` (see Section 15.3 [Library Search], page 211) when looking for Lisp files. If *filename* is absolute, *path* has no effect, but the suffixes in *suffixes* are still tried.

The optional argument *predicate*, if non-`nil`, specifies a predicate function for testing whether a candidate file is suitable. The predicate is passed the candidate file name as its single argument. If *predicate* is `nil` or omitted, `locate-file` uses `file-readable-p` as the predicate. See Section 25.6.2 [Kinds of Files], page 473, for other useful predicates, e.g. `file-executable-p` and `file-directory-p`.

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 absolute file 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` (see Section 37.1 [Subprocess Creation], page 257, vol. 2).

25.7 Changing File Names and Attributes

The functions in this section rename, copy, delete, link, and set the modes (permissions) 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 475.

`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 preserve-uid-gid preserve-selinux* [Command]

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.

If the optional argument *preserve-selinux* is non-`nil`, and Emacs has been compiled with SELinux support, this function attempts to copy the file's SELinux context (see Section 25.6.4 [File Attributes], page 475).

`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* **&optional** *trash* [Command]

This command deletes the file *filename*. If the file has multiple names, it continues to exist under the other names. If *filename* is a symbolic link, `delete-file` deletes only the symbolic link and not its target (though it does follow symbolic links at all levels of parent directories).

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 the optional argument *trash* is non-`nil` and the variable `delete-by-moving-to-trash` is non-`nil`, this command moves the file into the system Trash instead of deleting it. See Section "Miscellaneous File Operations" in *The GNU Emacs Manual*. When called interactively, *trash* is `t` if no prefix argument is given, and `nil` otherwise.

See also `delete-directory` in Section 25.10 [Create/Delete Dirs], page 493.

`set-file-modes` *filename mode* [Command]

This function sets the *file mode* (or *file permissions*) of *filename* to *mode*. It recursively follows symbolic links at all levels for *filename*.

If called non-interactively, *mode* must be an integer. Only the lowest 12 bits of the integer are used; on most systems, only the lowest 9 bits are meaningful. You can use the Lisp construct for octal numbers to enter *mode*. For example,

```
(set-file-modes #o644)
```

specifies that the file should be readable and writable for its owner, readable for group members, and readable for all other users. See Section "File Permissions" in *The* GNU `Coreutils` *Manual*, for a description of mode bit specifications.

Interactively, *mode* is read from the minibuffer using `read-file-modes` (see below), which lets the user type in either an integer or a string representing the permissions symbolically.

See Section 25.6.4 [File Attributes], page 475, for the function `file-modes`, which returns the permissions of a file.

`set-default-file-modes` *mode* [Function]

This function sets the default file permissions for new files created by Emacs and its subprocesses. Every file created with Emacs initially has these permissions, or a subset of them (`write-region` will not grant execute permissions even if the default file permissions allow execution). On Unix and GNU/Linux, the default permissions are given by the bitwise complement of the "umask" value.

The argument *mode* should be an integer which specifies the permissions, similar to `set-file-modes` above. Only the lowest 9 bits are meaningful.

The default file permissions have no effect when you save a modified version of an existing file; saving a file preserves its existing permissions.

`default-file-modes` [Function]

This function returns the default file permissions, as an integer.

`read-file-modes` **&optional** *prompt base-file* [Function]

This function reads a set of file mode bits from the minibuffer. The first optional argument *prompt* specifies a non-default prompt. Second second optional argument *base-file* is the name of a file on whose permissions to base the mode bits that this function returns, if what the user types specifies mode bits relative to permissions of an existing file.

If user input represents an octal number, this function returns that number. If it is a complete symbolic specification of mode bits, as in `"u=rwx"`, the function converts it to the equivalent numeric value using `file-modes-symbolic-to-number` and returns the result. If the specification is relative, as in `"o+g"`, then the permissions on which the specification is based are taken from the mode bits of *base-file*. If *base-file* is omitted or `nil`, the function uses `0` as the base mode bits. The complete and relative specifications can be combined, as in `"u+r,g+rx,o+r,g-w"`. See Section "File Permissions" in *The* GNU `Coreutils` *Manual*, for a description of file mode specifications.

`file-modes-symbolic-to-number` *modes* **&optional** *base-modes* [Function]

This function converts a symbolic file mode specification in *modes* into the equivalent integer value. If the symbolic specification is based on an existing file, that file's mode bits are taken from the optional argument *base-modes*; if that argument is omitted or `nil`, it defaults to 0, i.e. no access rights at all.

`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 399, vol. 2).

`set-file-selinux-context` *filename context* [Function]

This function sets the SELinux security context of the file *filename* to *context*. See Section 25.6.4 [File Attributes], page 475, for a brief description of SELinux contexts. The *context* argument should be a list `(`*`user role type range`*`)`, like the return value of `file-selinux-context`. The function does nothing if SELinux is disabled, or if Emacs was compiled without SELinux support.

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. 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.

For some purposes, the nondirectory part is further subdivided into the name proper and the *version number*. On most systems, only backup files have version numbers in their names.

`file-name-directory` *filename* [Function]

This function returns the directory part of *filename*, as a directory name (see Section 25.8.3 [Directory Names], page 485), 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.

```
(file-name-directory "lewis/foo")  ; Unix example
     ⇒ "lewis/"
(file-name-directory "foo")        ; Unix example
     ⇒ nil
```

`file-name-nondirectory` *filename* [Function]

This function returns the nondirectory part of *filename*.

```
(file-name-nondirectory "lewis/foo")
     ⇒ "foo"
(file-name-nondirectory "foo")
     ⇒ "foo"
(file-name-nondirectory "lewis/")
     ⇒ ""
```

`file-name-sans-versions` *filename* **&optional** *keep-backup-version* [Function]

This function returns *filename* with any file version numbers, backup version numbers, or trailing tildes discarded.

If *keep-backup-version* is non-`nil`, then true file version numbers understood as such by the file system are discarded from the return value, but backup version numbers are kept.

```
(file-name-sans-versions "~rms/foo.~1~")
     ⇒ "~rms/foo"
(file-name-sans-versions "~rms/foo~")
     ⇒ "~rms/foo"
(file-name-sans-versions "~rms/foo")
     ⇒ "~rms/foo"
```

`file-name-extension` *filename* **&optional** *period* [Function]

This function returns *filename*'s final "extension", if any, after applying `file-name-sans-versions` to remove any version/backup part. The extension, in a file name, is the part that follows the last '`.`' in the last name component (minus any version/backup part).

This function returns `nil` for extensionless file names such as '`foo`'. It returns `""` for null extensions, as in '`foo.`'. If the last component of a file name begins with a '`.`', that '`.`' doesn't count as the beginning of an extension. Thus, '`.emacs`''s "extension" is `nil`, not '`.emacs`'.

If *period* is non-`nil`, then the returned value includes the period that delimits the extension, and if *filename* has no extension, the value is `""`.

`file-name-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 '`/`' or a '`~`' (see [abbreviate-file-name], page 485), and a relative one does not. On MS-DOS and MS-Windows, an absolute file name starts with a slash or a backslash, or with a drive specification '`x:/`', where *x* is the *drive letter*.

`file-name-absolute-p` *filename* [Function]

This function returns `t` if file *filename* is an absolute file name, `nil` otherwise.

```
     (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 486). 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 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).

```
(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.

```
(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`.

To convert a directory name to its abbreviation, use this function:

`abbreviate-file-name` *filename* [Function]

This function returns an abbreviated form of *filename*. It applies the abbreviations specified in `directory-abbrev-alist` (see Section "File Aliases" in *The GNU Emacs*

Manual), then substitutes '~' for the user's home directory if the argument names a file in the home directory or one of its subdirectories. If the home directory is a root directory, it is not replaced with '~', because this does not make the result shorter on many systems.

You can use this function for directory names and for file names, because it recognizes abbreviations even as part of the name.

25.8.4 Functions that Expand Filenames

Expanding a file name means converting a relative file name to an absolute one. Since this is done relative to a default directory, you must specify the default directory name as well as the file name to be expanded. It also involves expanding abbreviations like '`~/`' and eliminating redundancies like '`./`' and '*`name`*`/../`'.

`expand-file-name` *filename* **&optional** *directory* [Function]

This function converts *filename* to an absolute file name. If *directory* is supplied, it is the default directory to start with if *filename* is relative. (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 474.

`default-directory` [Variable]

The value of this buffer-local variable is the default directory for the current buffer. It should be an absolute directory name; it may start with '`~`'. This variable is buffer-local in every buffer.

`expand-file-name` uses the default directory when its second argument is `nil`.

The value is always a string ending with a slash.

```
default-directory
     ⇒ "/user/lewis/manual/"
```

`substitute-in-file-name` *filename* [Function]

This function replaces environment variable references in *filename* with the environment variable values. Following standard Unix shell syntax, '`$`' is the prefix to substitute an environment variable value. If the input contains '`$$`', that is converted to '`$`'; this gives the user a way to "quote" a '`$`'.

The environment variable name is the series of alphanumeric characters (including underscores) that follow the '`$`'. If the character following the '`$`' is a '`{`', then the variable name is everything up to the matching '`}`'.

Calling `substitute-in-file-name` on output produced by `substitute-in-file-name` tends to give incorrect results. For instance, use of '`$$`' to quote a single '`$`' won't work properly, and '`$`' in an environment variable's value could lead to repeated substitution. Therefore, programs that call this function and put the output where it will be passed to this function need to double all '`$`' characters to prevent subsequent incorrect results.

Here we assume that the environment variable `HOME`, which holds the user's home directory 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.
```

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 485.

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`.

`temporary-file-directory` [User Option]

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 485), but it is good for Lisp programs to cope if the value is a directory's file name instead. Using the value as the second argument to `expand-file-name` is a good way to achieve that.

The default value is determined in a reasonable way for your operating system; it is based on the `TMPDIR`, `TMP` and `TEMP` environment variables, with a fall-back to a system-dependent name if none of these variables is defined.

Even if you do not use `make-temp-file` to create the temporary file, you should still use this variable to decide which directory to put the file in. However, if you expect the file to be small, you should use `small-temporary-file-directory` first if that is non-`nil`.

`small-temporary-file-directory` [User Option]

This variable specifies the directory name for creating certain temporary files, which are likely to be small.

If you want to write a temporary file which is likely to be small, you should compute the directory like this:

```
(make-temp-file
  (expand-file-name prefix
                    (or small-temporary-file-directory
                        temporary-file-directory)))
```

`make-temp-name` *base-name* [Function]

This function generates a string that can be used as a unique file name. The name starts with *base-name*, and has several random characters appended to it, which are different in each Emacs job. It is like `make-temp-file` except that (i) it just constructs a name, and does not create a file, and (ii) *base-name* should be an absolute file name (on MS-DOS, this function can truncate *base-name* to fit into the 8+3 file-name limits).

Warning: In most cases, you should not use this function; use `make-temp-file` instead! This function is susceptible to a race condition, between the `make-temp-name` call and the creation of the file, which in some cases may cause a security hole.

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 300.

`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

Sometimes, an Emacs Lisp program needs to specify a standard file name for a particular use—typically, to hold configuration data specified by the current user. Usually, such files should be located in the directory specified by `user-emacs-directory`, which is ‘`~/.emacs.d`’ by default (see Section 39.1.2 [Init File], page 389, vol. 2). For example, abbrev definitions are stored by default in ‘`~/.emacs.d/abbrev_defs`’. The easiest way to specify such a file name is to use the function `locate-user-emacs-file`.

`locate-user-emacs-file` *base-name* **&optional** *old-name* [Function]

This function returns an absolute file name for an Emacs-specific configuration or data file. The argument ‘`base-name`’ should be a relative file name. The return value is the absolute name of a file in the directory specified by `user-emacs-directory`; if that directory does not exist, this function creates it.

If the optional argument *old-name* is non-`nil`, it specifies a file in the user’s home directory, ‘`~/old-name`’. If such a file exists, the return value is the absolute name of that file, instead of the file specified by *base-name*. This argument is intended to be used by Emacs packages to provide backward compatibility. For instance, prior to the introduction of `user-emacs-directory`, the abbrev file was located in '`~/.abbrev_defs`'. Here is the definition of `abbrev-file-name`:

```
(defcustom abbrev-file-name
  (locate-user-emacs-file "abbrev_defs" ".abbrev_defs")
  "Default name of file from which to read abbrevs."
  ...
  :type 'file)
```

A lower-level function for standardizing file names, which `locate-user-emacs-file` uses as a subroutine, is `convert-standard-filename`.

`convert-standard-filename` *filename* [Function]

This function returns a file name based on *filename*, which fits the conventions of the current operating system.

On GNU and Unix systems, this simply returns *filename*. On other operating systems, it may enforce system-specific file name conventions; for example, on MS-DOS this function performs a variety of changes to enforce MS-DOS file name limitations, including converting any leading '`.`' to '`_`' and truncating to three characters after the '`.`'.

The recommended way to use this function is to specify a name which fits the conventions of GNU and Unix systems, and pass it to `convert-standard-filename`.

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 match-regexp nosort id-format* [Function]

This is similar to `directory-files` in deciding which files to report on and how to report their names. However, instead of returning a list of file names, it returns for each file a list `(`*`filename`*` . `*`attributes`*`)`, where *attributes* is what `file-attributes` 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 476).

`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, Copying and Deleting Directories

Most Emacs Lisp file-manipulation functions get errors when used on files that are directories. For example, you cannot delete a directory with `delete-file`. These special functions exist to create and delete directories.

`make-directory` *dirname* **&optional** *parents* [Command]

This command creates a directory named *dirname*. If *parents* is non-`nil`, as is always the case in an interactive call, that means to create the parent directories first, if they don't already exist.

`mkdir` is an alias for this.

`copy-directory` *dirname newname* **&optional** *keep-time parents copy-contents* [Command]

This command copies the directory named *dirname* to *newname*. If *newname* names an existing directory, *dirname* will be copied to a subdirectory there.

It always sets the file modes of the copied files to match the corresponding original file.

The third argument *keep-time* non-`nil` means to preserve the modification time of the copied files. A prefix arg makes *keep-time* non-`nil`.

The fourth argument *parents* says whether to create parent directories if they don't exist. Interactively, this happens by default.

The fifth argument *copy-contents*, if non-`nil`, means to copy the contents of *dirname* directly into *newname* if the latter is an existing directory, instead of copying *dirname* into it as a subdirectory.

`delete-directory` *dirname* **&optional** *recursive trash* [Command]

This command deletes the directory named *dirname*. The function `delete-file` does not work for files that are directories; you must use `delete-directory` for them. If *recursive* is `nil`, and the directory contains any files, `delete-directory` signals an error.

`delete-directory` only follows symbolic links at the level of parent directories.

If the optional argument *trash* is non-`nil` and the variable `delete-by-moving-to-trash` is non-`nil`, this command moves the file into the system Trash instead of deleting it. See Section "Miscellaneous File Operations" in *The GNU Emacs Manual*. When called interactively, *trash* is `t` if no prefix argument is given, and `nil` otherwise.

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 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-directory`, `copy-file`,
`delete-directory`, `delete-file`,
`diff-latest-backup-file`,
`directory-file-name`,
`directory-files`,
`directory-files-and-attributes`,
`dired-compress-file`, `dired-uncache`,
`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`,
`get-file-buffer`,
`insert-directory`,
`insert-file-contents`,
`load`, `make-directory`,
`make-directory-internal`,
`make-symbolic-link`,
`process-file`,
`rename-file`, `set-file-modes`,
`set-visited-file-modtime`, `shell-command`,
`start-file-process`,
`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* **&optional** *identification connected* [Function]

This function tests whether *filename* is a remote file. If *filename* is local (not remote), the return value is `nil`. If *filename* is indeed remote, the return value is a string that identifies the remote system.

This identifier string can include a host name and a user name, as well as characters designating the method used to access the remote system. For example, the remote identifier string for the filename `/sudo::/some/file` is `/sudo:root@localhost:`.

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. Implementers of file handlers need to ensure this principle is valid.

identification specifies which part of the identifier shall be returned as string. *identification* can be the symbol `method`, `user` or `host`; any other value is handled like `nil` and means to return the complete identifier string. In the example above, the remote `user` identifier string would be `root`.

If *connected* is non-`nil`, this function returns `nil` even if *filename* is remote, if Emacs has no network connection to its host. This is useful when you want to avoid the delay of making connections when they don't exist.

`unhandled-file-name-directory` *filename* [Function]

This function returns the name of a directory that is not magic. 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. If *filename* is not accessible from a local process, then the file name handler should indicate it by returning `nil`.

This is useful for running a subprocess; every subprocess must have a non-magic directory to serve as its current directory, and this function is a good way to come up with one.

`remote-file-name-inhibit-cache` [User Option]

The attributes of remote files can be cached for better performance. If they are changed outside of Emacs's control, the cached values become invalid, and must be reread.

When this variable is set to `nil`, cached values are never expired. Use this setting with caution, only if you are sure nothing other than Emacs ever changes the remote files. If it is set to `t`, cached values are never used. This is the safest value, but could result in performance degradation.

A compromise is to set it to a positive number. This means that cached values are used for that amount of seconds since they were cached. If a remote file is checked regularly, it might be a good idea to let-bind this variable to a value less than the time period between consecutive checks. For example:

```
(defun display-time-file-nonempty-p (file)
  (let ((remote-file-name-inhibit-cache
         (- display-time-interval 5)))
    (and (file-exists-p file)
         (< 0 (nth 7 (file-attributes
                      (file-chase-links file)))))))
```

25.12 File Format Conversion

Emacs performs several steps to convert the data in a buffer (text, text properties, and possibly other information) to and from a representation suitable for storing into a file. This section describes the fundamental functions that perform this *format conversion*, namely `insert-file-contents` for reading a file into a buffer, and `write-region` for writing a buffer into a file.

25.12.1 Overview

The function `insert-file-contents`:

- initially, inserts bytes from the file into the buffer;

- decodes bytes to characters as appropriate;
- processes formats as defined by entries in `format-alist`; and
- calls functions in `after-insert-file-functions`.

The function `write-region`:

- initially, calls functions in `write-region-annotate-functions`;
- processes formats as defined by entries in `format-alist`;
- encodes characters to bytes as appropriate; and
- modifies the file with the bytes.

This shows the symmetry of the lowest-level operations; reading and writing handle things in opposite order. The rest of this section describes the two facilities surrounding the three variables named above, as well as some related functions. Section 33.9 [Coding Systems], page 193, vol. 2, for details on character encoding and decoding.

25.12.2 Round-Trip Specification

The most general of the two facilities is controlled by the variable `format-alist`, a list of *file format* specifications, which describe textual representations used in files for the data in an Emacs buffer. The descriptions for reading and writing are paired, which is why we call this "round-trip" specification (see Section 25.12.3 [Format Conversion Piecemeal], page 500, for non-paired specification).

`format-alist` [Variable]

This list contains one format definition for each defined file format. Each format definition is a list of this form:

```
(name doc-string regexp from-fn to-fn modify mode-fn preserve)
```

Here is what the elements in a format definition mean:

name The name of this format.

doc-string A documentation string for the format.

regexp A regular expression which is used to recognize files represented in this format. If `nil`, the format is never applied automatically.

from-fn A shell command or function to decode data in this format (to convert file data into the usual Emacs data representation).

A shell command is represented as a string; Emacs runs the command as a filter to perform the conversion.

If *from-fn* is a function, it is called with two arguments, *begin* and *end*, which specify the part of the buffer it should convert. It should convert the text by editing it in place. Since this can change the length of the text, *from-fn* should return the modified end position.

One responsibility of *from-fn* is to make sure that the beginning of the file no longer matches *regexp*. Otherwise it is likely to get called again.

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.

preserve A flag, `t` if `format-write-file` should not remove this format from `buffer-file-format`.

The function `insert-file-contents` automatically recognizes file formats when it reads the specified file. It checks the text of the beginning of the file against the regular expressions of the format definitions, and if it finds a match, it calls the decoding function for that format. Then it checks all the known formats over again. It keeps checking them until none of them is applicable.

Visiting a file, with `find-file-noselect` or the commands that use it, performs conversion likewise (because it calls `insert-file-contents`); it also calls the mode function for each format that it decodes. It stores a list of the format names in the buffer-local variable `buffer-file-format`.

`buffer-file-format` [Variable]

This variable states the format of the visited file. More precisely, this is a list of the file format names that were decoded in the course of visiting the current buffer's file. It is always buffer-local in all buffers.

When `write-region` writes data into a file, it first calls the encoding functions for the formats listed in `buffer-file-format`, in the order of appearance in the list.

`format-write-file` *file format* **&optional** *confirm* [Command]

This command writes the current buffer contents into the file *file* in a format based on *format*, which is a list of format names. It constructs the actual format starting from *format*, then appending any elements from the value of `buffer-file-format` with a non-`nil` *preserve* flag (see above), if they are not already present in *format*. It then updates `buffer-file-format` with this format, making it the default for future saves. Except for the *format* argument, this command is similar to `write-file`. In particular, *confirm* has the same meaning and interactive treatment as the corresponding argument to `write-file`. See [Definition of write-file], page 465.

`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 467).

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.

25.12.3 Piecemeal Specification

In contrast to the round-trip specification described in the previous subsection (see Section 25.12.2 [Format Conversion Round-Trip], page 498), you can use the variables `after-insert-file-functions` and `write-region-annotate-functions` to separately control the respective reading and writing conversions.

Conversion starts with one representation and produces another representation. When there is only one conversion to do, there is no conflict about what to start with. However, when there are multiple conversions involved, conflict may arise when two conversions need to start with the same data.

This situation is best understood in the context of converting text properties during `write-region`. For example, the character at position 42 in a buffer is '`X`' with a text property `foo`. If the conversion for `foo` is done by inserting into the buffer, say, '`FOO:`', then that changes the character at position 42 from '`X`' to '`F`'. The next conversion will start with the wrong data straight away.

To avoid conflict, cooperative conversions do not modify the buffer, but instead specify *annotations*, a list of elements of the form `(`*`position`*` . `*`string`*`)`, sorted in order of increasing *position*.

If there is more than one conversion, `write-region` merges their annotations destructively into one sorted list. Later, when the text from the buffer is actually written to the file, it intermixes the specified annotations at the corresponding positions. All this takes place without modifying the buffer.

In contrast, when reading, the annotations intermixed with the text are handled immediately. `insert-file-contents` sets point to the beginning of some text to be converted, then calls the conversion functions with the length of that text. These functions should always return with point at the beginning of the inserted text. This approach makes sense for reading because annotations removed by the first converter can't be mistakenly processed by a later converter. Each conversion function should scan for the annotations it recognizes, remove the annotation, modify the buffer text (to set a text property, for example), and return the updated length of the text, as it stands after those changes. The value returned by one function becomes the argument to the next function.

`write-region-annotate-functions` [Variable]

> A list of functions for `write-region` to call. Each function in the list is called with two arguments: the start and end of the region to be written. These functions should not alter the contents of the buffer. Instead, they should return annotations.
>
> As a special case, a function may return with a different buffer current. Emacs takes this to mean that the current buffer contains altered text to be output. It therefore changes the *start* and *end* arguments of the `write-region` call, giving them the values of `point-min` and `point-max` in the new buffer, respectively. It also discards all previous annotations, because they should have been dealt with by this function.

`write-region-post-annotation-function` [Variable]

> The value of this variable, if non-`nil`, should be a function. This function is called, with no arguments, after `write-region` has completed.
>
> If any function in `write-region-annotate-functions` returns with a different buffer current, Emacs calls `write-region-post-annotation-function` more than once. Emacs calls it with the last buffer that was current, and again with the buffer before that, and so on back to the original buffer.
>
> Thus, a function in `write-region-annotate-functions` can create a buffer, give this variable the local value of `kill-buffer` in that buffer, set up the buffer with altered text, and make the buffer current. The buffer will be killed after `write-region` is done.

`after-insert-file-functions` [Variable]

> Each function in this list is called by `insert-file-contents` with one argument, the number of characters inserted, and with point at the beginning of the inserted text. Each function should leave point unchanged, and return the new character count describing the inserted text as modified by the function.

We invite users to write Lisp programs to store and retrieve text properties in files, using these hooks, and thus to experiment with various data formats and find good ones. Eventually we hope users will produce good, general extensions we can install in Emacs.

We suggest not trying to handle arbitrary Lisp objects as text property names or values—because a program that general is probably difficult to write, and slow. Instead, choose a set of possible data types that are reasonably flexible, and not too hard to encode.

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 context 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 475), *context* is a list describing the original file's SELinux context (see Section 25.6.4 [File Attributes], page 475), 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
          (lambda ()
            (set (make-local-variable 'make-backup-files) nil)))
```

`backup-enable-predicate` [Variable]

This variable's value is a function to be called on certain occasions to decide whether a file should have backup files. The function receives one argument, an absolute file name to consider. If the function returns `nil`, backups are disabled for that file. Otherwise, the other variables in this section say whether and how to make backups.

The default value is `normal-backup-enable-predicate`, which checks for files in `temporary-file-directory` and `small-temporary-file-directory`.

`backup-inhibited` [Variable]

If this variable is non-`nil`, backups are inhibited. It records the result of testing `backup-enable-predicate` on the visited file name. It can also coherently be used by other mechanisms that inhibit backups based on which file is visited. For example, VC sets this variable non-`nil` to prevent making backups for files managed with a version control system.

This is a permanent local, so that changing the major mode does not lose its value. Major modes should not set this variable—they should set `make-backup-files` instead.

`backup-directory-alist` [User Option]

This variable's value is an alist of filename patterns and backup 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` (the default), or it fails to match a filename, the backup is made in the original file's directory.

On MS-DOS filesystems without long names this variable is always ignored.

`make-backup-file-name-function` [User Option]

This variable's value is a function to use for making 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 505.

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 465.

`backup-by-copying` [User Option]
: If this variable is non-`nil`, Emacs always makes backup files by copying. The default is `nil`.

The following three variables, when non-`nil`, cause the second method to be used in certain special cases. They have no effect on the treatment of files that don't fall into the special cases.

`backup-by-copying-when-linked` [User Option]
: If this variable is non-`nil`, Emacs makes backups by copying for files with multiple names (hard links). The default is `nil`.

  This variable is significant only if `backup-by-copying` is `nil`, since copying is always used when that variable is non-`nil`.

`backup-by-copying-when-mismatch` [User Option]
: If this variable is non-`nil` (the default), Emacs makes backups by copying in cases where renaming would change either the owner or the group of the file.

  The value has no effect when renaming would not alter the owner or group of the file; that is, for files which are owned by the user and whose group matches the default for a new file created there by the user.

  This variable is significant only if `backup-by-copying` is `nil`, since copying is always used when that variable is non-`nil`.

`backup-by-copying-when-privileged-mismatch` [User Option]
: This variable, if non-`nil`, specifies the same behavior as `backup-by-copying-when-mismatch`, but only for certain user-id 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 505) 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 505.

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. When called from Lisp with no argument, it turns auto-saving on. With an argument *arg*, it 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.

If it is −2, that means this buffer should disregard changes in buffer size; in particular, it should not shut off auto-saving temporarily due to changes in buffer size.

`auto-save-list-file-name` [Variable]

This variable (if non-`nil`) specifies a file for recording the names of all the auto-save files. Each time Emacs does auto-saving, it writes two lines into this file for each buffer that has auto-saving enabled. The first line gives the name of the visited file (it's empty if the buffer has none), and the second gives the name of the auto-save file.

When Emacs exits normally, it deletes this file; if Emacs crashes, you can look in the file to find all the auto-save files that might contain work that was otherwise lost. The `recover-session` command uses this file to find them.

The default name for this file specifies your home directory and starts with '`.saves-`'. It also contains the Emacs process ID and the host name.

`auto-save-list-file-prefix` [User Option]

After Emacs reads your init file, it initializes `auto-save-list-file-name` (if you have not already set it non-`nil`) based on this prefix, adding the host name and process ID. If you set this to `nil` in your init file, then Emacs does not initialize `auto-save-list-file-name`.

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.

`revert-buffer-in-progress-p` [Variable]
: `revert-buffer` binds this variable to a non-`nil` value while it is working.

You can customize how `revert-buffer` does its work by setting the variables described in the rest of this section.

`revert-without-query` [User Option]
: This variable holds a list of files that should be reverted without query. The value is a list of regular expressions. If the visited file name matches one of these regular expressions, and the file has changed on disk but the buffer is not modified, then `revert-buffer` reverts the file without asking the user for confirmation.

Some major modes customize `revert-buffer` by making buffer-local bindings for these variables:

`revert-buffer-function` [Variable]
: The value of this variable is the function to use to revert this buffer. 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`.

`buffer-stale-function` [Variable]
: The value of this variable, if non-`nil`, specifies a function to call to check whether a non-file buffer needs reverting (see Section "Supporting additional buffers" in *Specialized Emacs Features*).

Index